香港中文大學研究院教育學部

THE CHINESE UNIVERSITY OF HONG KONG

GRADUATE SCHOOL • DIVISION OF EDUCATION

文科教育碩士論文

Master of Arts in Education Thesis

論文題目
Thesis Title

Effects of Language Features, Templates, and Procedural Skills on Problem Solving in Programming

語法、知統及程序技巧對程式解難能力的影響

撰作語言
Language Used

英文
English

研究生姓名
Name of Student

江紹祥
Kong Siu Cheung

專修範圍
Specialization

教育測量與評鑑
Educational Measurement and Evaluation

論文考試委員會

Thesis Examination Committee

論文導師 **Thesis Supervisor**	鍾財文先生	Mr. Choi-man CHUNG
校內委員 **Internal Examiner**	麥思源博士	Dr. Se-yuen MAK
校內委員 **Internal Examiner**	夏偉富博士	Dr. Rex HEYWORTH
校外委員 **External Examiner**	譚添鉅博士	Dr. Peter T.K. TAM
學部主任 **Division Head**	蕭炳基博士	Dr. Ping-kee SIU

論文通過日期
Date of Approval

August 17, 1988

Effects of Language Features, Templates, and Procedural Skills on Problem Solving in Programming

by

Kong Siu Cheung
under the supervision of
Mr. Chung Choi Man

A Thesis Submitted to The
School of Education
The Chinese University of Hong Kong
in Partial Fulfilments of
The Requirements for The Degree of
Master of Arts in Education

June, 1988

ACKNOWLEDGEMENTS

I would especially like to express my indebtedness to Mr C.M. Chung, my thesis supervisor, for his most helpful insight, guidance, encouragement, support and invaluable comments in every stage of the thesis development. I am also indebted to Dr. Rex Heyworth and Dr. S.Y. Mak for their invaluable comments and suggestions on the thesis proposal. Without the advice of the thesis committee members, this thesis could never be able to complete.

The author would also like to express his gratitude to Mr T.W. Lum, the principal of Notre Dame college, for his advice on the instruments developed in this research and his generosity in providing the physical environment and students for conducting the pilot study. The author would also wish to take this opportunity to acknowledge Mr H.Y. Lee, Mr H.M. Leung, Mr P.H. Tam, and Miss Betty Key for helping in refining the instruments in this research and assistance in identifying schools to participate in and providing data for this research. The author would also like to take this opportunity to thank all the computer studies teachers and students of the participating schools in this research.

I have to express my special thanks and appreciation to my wife who not only quietly supported me in numerous ways throughout this thesis development but also assisted me technically in data coding and data entry. I apologize for occupying so much of our leisure time in writing the thesis. To my parents, I have to thank for their consistent whole-heartedly spiritual support throughout the years.

ABSTRACT

The popularity of computer programming in pre-college students and the assumption that activities engaged in it would foster general problem solving skills arose the interest of the researcher to investigate the contents in programming course. The present research purports to study three aspects of programming. First, to study what componential knowledge and skills are related to programming. Second, to investigate how these knowledge and skills are affecting programming. This study was based on Linn's theoretical postulation on the chain of cognitive accomplishment for programming. Third, to study what individual difference variables are significant to programming performance.

Literature review indicated that knowledge including features of a programming language and templates abstracted from programming experiences are related to programming. Template is defined as stereotypic patterns of code perform commonly encountered tasks in programming. Literature discussing the process of problem solving in programming revealed that procedural skills including planning a program, reformulating the existed program and testing the finished program are related to programming. Four instruments were developed in this research to measure students' language proficiency, template possession, procedural skills and problem solving ability in programming. Two hundred and sixty form five students in Hong Kong were selected to take the tests. Three interviews on studying programming behavior were also conducted to supplement the quantitative analysis.

The results from the correlational study indicated that

knowledge including language features and templates; and procedural skills including planning, testing and reformulating were significant components related to programming. Results from the protocol and causal path model analyses revealed that proficiency in the features of a programming language was a necessary but not a sufficient condition in programming. Proficiency in the features of a programming language underpinned programming performance through the mediating effects of template abstraction and the general sophistication with the procedural skills. Planning skill was found most prominent in problem solving in programming. Results also revealed that template possession facilitated planning skill, therefore abstracting knowledge of template may bridge the gap between syntax learning and problem solving in programming. In this research, ability was the individual difference variable that was most strongly related to student performance in all programming tasks. Males and females did about equally well generally with males slightly outscoring females on some programming tasks. Students in low teacher-student ratio programming classes slightly outperformed their counterparts. Students did not possess home computer performed equally well as those who possessed in all programming tasks.

Results in this research indicated that curriculum and method of instrution in programming education should be reconciled to the findings in this research in order to maximize the cognitive outcomes form such instruction. Explicit instruction on template and procedural skills seems necessary to foster problem solving skill in programming.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I INTRODUCTION

1.1 BACKGROUND AND PROBLEM OF THE STUDY

Problem solving has been emphasized as the central theme of education for many years. It is generally regarded as the highest cognitive skill desirable to be acquired by learners in their educational process (Gagne, 1985; Scandura, 1977; Newell & Simon, 1972).

Claims For Cognitive Effects Of Learning To Program

Since programming is a complex problem solving activity performed by professional programmers, engaging in such an activity has been perceived by psychologists, computer scientists, and educators as a powerful means for enhancing thinking and the development of good problem solving skills in children. Such an assumption on the cognitive consequences of programming is advocated in Papert's book, Mindstorms (Papert, 1980). Papert predicted that children can learn good problem solving skills by submerging them in a programming environment, like LOGO programming. This situation is just like a foreigner learning French in France. Papert believes that children can learn programming through "a process that takes place without deliberate or organized teaching" (Papert, 1980, p.8) in the LOGO programming environment.

Does learning to program computers have a broad impact on children's thinking ability (Salomon & Perkins, 1987) ? The implications of this question have been lucidly summarized by Pea and Kurland (Pea & Kurland, 1987). They comment on the wide-spread belief that

"This belief, although new in its application to this domain, is an old idea in a new costume which has been worn often before. In its common extreme form, it is based on an assumption about learning - that spontaneous experience with powerful symbolic system will have beneficial cognitive consequences, especially for higher order cognitive skills. Similiar arguments have been offered in centuries past for mathematics, logic, writing systems and Latin."

The next important step is to clarify the assumptions by empirical testing. Many attempts have been made to evaluate the cognitive consequences of learning to program. This work is best exemplified by the assessments conducted at the Bank Street College of Education (Pea & Kurland, 1984; Kurland, Mawby, & Cahir, 1984; Pea, 1983) and Lawrence Hall of Science (Dalbey, Tourniaire, & Linn, 1986). Pea and Kurland summarized that there is little evidence for these claims, at least, at the present moment (Pea & Kurland, 1987). Johanson commented that the principle weakness of research on the cognitive consequences of programming instruction most probably has been its inadequate consideration of curricular issues (Johanson, 1988). A more fundamental research direction is called for, how and what cognitive componential skills are involved in programming (Allwood, 1986). Findings from such studies are essential to carry on empirical studies on the cognitive effects on programming.

Content-free Versus Knowledge-based Problem Solving

The researches conducted by the information-processing theorists in the seventy's mainly focused on problem solving strategies involved in content-free problems such as the missionaries and cannibals problems (e.g. Reed, Ernst, & Banerji, 1974). In the past 10 years, problem solving research has under-gone a transformation switching from investigating artificial puzzles to the more semantically rich textbook problems in physics or mathematics (Gick, 1986). Such a change in orientation facilitates the possibility to apply the research findings to the classroom problem solving activities. More fundamentally, recent research findings in the cognitive psychology field do indicate that well-structured knowledge in a particular domain is essential in solving problems in it (Glaser, 1984). This is one of the background of the present study to investigate problem solving in the computer programming domain.

The Popularity Of Computer Programming

Though computer programming was limited to a small population of professional programmers in the past several decades, the drastic decrease in computer hardware cost and the tremendous increase in their capabilities provided the foundation for the wide application of computer in the society and specially in education.

The introduction of microcomputer in recent years in secondary schools has brought-up the popularity of computer programming in responding to the wide application of the machine all round the world (IFIP working conference, 1984).

Each year, there are several million pre-college age children in the U.S.A. receiving instruction in computer programming. France has recently made programming compulsory in their pre-college curriculum (Pea & Kurland, 1987). The following table shows the number of candidates taking computer studies in public examination in recent years in Hong Kong (Shin, 1987).

Table 1:No. of Candidates taking computer studies in Hong Kong

Year	Number of candidates
1984	2561
1985	4145
1986	6379
1987	around 10000

One of the main objectives in computer studies curriculum in secondary school is to enhance students' problem solving skills through programming. In Hong Kong, each student spends about 80 hours of programming in the course which account for nearly half of the syllabus coverage. Additional hours spending on programming assignments after school are numerous and are difficult to estimate. Such a large number of learners engaged in programming for such a long time deserves studying. The present study may serve to find ways to maximize the cognitive outcomes from such instruction.

1.2 Purpose Of The Study

The present study purports to investigate three aspects of programming, namely, what componential knowledge and skills are involved, how are these knowledge and skills affecting solvers' performance and what are the individual differences that affect the programming performance.

Programming Knowledge And Skills

There are two predominant tendencies in thinking about learning to program today. The first belief follows the behaviorist tradition viewing learning programming simply as an accumulation of "facts". Learning to program is to learn vocabulary of primitives and syntactic rules for constructing acceptable arrangements of commands. This belief underlies most programming instruction. Learning programming becomes a rote memorization of language features in a programming language (Shneiderman & Mayer, 1979).

In reaction to the first belief, a contrasting belief is that through learning to programming, children are learning not only porgramming but also acquiring powerfully general higher cognitive skills such as planning abilities, problem solving heuristics, testing and debugging skills (Papert, 1980).

The two beliefs are polar opposites and neither is acceptable. However, they reflect that programming requires both the familiarity of the features of a programming language and some design skills like planning, testing and reformulation (Linn, 1985). It had been reported that expert programmers took only 20-25 percent of the total time on

writing the actual code of a program. The rest of the time was spent on writing specifications, planning, designing algorithms, debugging and testing code (Kurland, Mawby & Cahir, 1984). However, empirical evidences indicate that pre-college student programmers seldom plan, test, and debug their programs systematically (Pintrich et al., 1987; Webb et al., 1986). Kreitzberg and Swanson (1974) in discussing an introductory programming curriculum had pointed out that the gap between knowledge of vocabulary and syntax and problem solution is too great for student programmers.

Knowledge of programming entailed not only just knowing the individual commands, but also an understanding of the relationship between commands and the rules of the language which determine the control structure of the program. Novices often cannot tell what a program does (Kurland, Clement, Mawby & Pea, 1984; Mayer, 1981) but experts always make use of their programming knowledge to make abstraction on what a program segment or program does. These abstracted programming knowledge will be cumulated as templates for later usage in pogramming.

To solve programming problems, novice programmers also need to acquire planning skills to break problems down into a much finer set of subtasks that are recognizable to them in terms of their programming knowledge (Perkins, Hancock, Hobbs, Martin, & Simmons, 1986). Skills involving setting of test data to fully test run a program and skills in reformulating the errors so found are postulated by Linn (1985) as essential skills in sussessful programming.

Since programming is a complex cognitive activity,

knowledge and skills facilitating its performance is difficult to be tested comprehensively. Knowledge including language features and templates, and skills involving planning, testing and reformulation will be investigated in this study to see which are essential in programming. The following figure summarizes the components postulated to be related to programming in the present study.

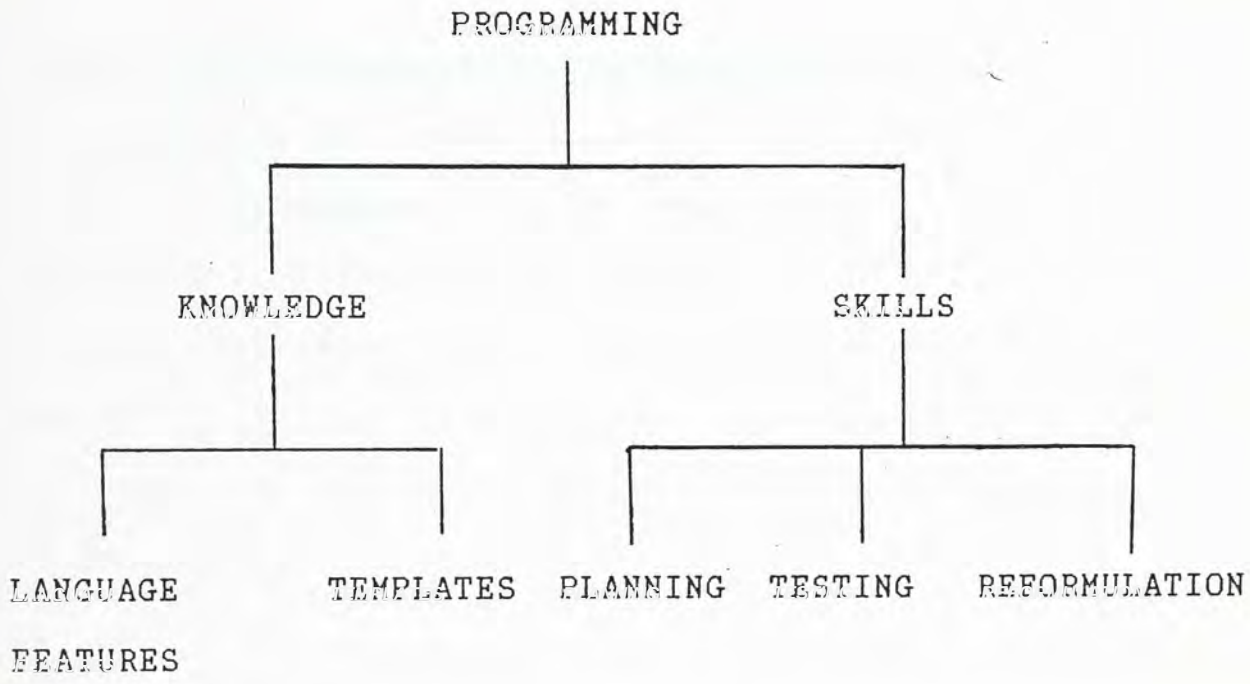


Figure 1 : Postulated componential knowledge and skills required in problem solving in programming

The Role Of Knowledge And Skills In Programming

The role of knowledge and skills in relation to problem solving in programming is lucidly summarized by Linn. Linn has identified a chain of cognitive accomplishment for programming that culminates in complex problem solving skills (Linn, 1985). The chain includes three links: language features, design skills, and generalizable problem solving

skills. The present study focuses on the first two links, namely, language features and design skills. Design skills in Linn's model include templates and procedural skills. Procedural skills include planning, testing and reformulating in programming. This study desires to explore how language features, templates and procedural skills are affecting problem solving in programming. If possible, a path analysis will be employed for the exploration.

Individual Differences On Programming Performances

Students do not react similarly to problem solving in computer programming. To what extent is programming performances a reflection of differences in the background of student ? This study will assess the role of background variables including (a) general ability (high/low), (b) gender, (c) home computer possession, and (d) teacher-student ratio (high/low).

1.3 SIGNIFICANCE OF THE STUDY

One of the significance of the present study is its impact on programming education. Finding out what knowledge and cognitive skills are likely in programming will be beneficial in designing a better programming curriculum. Figuring out how these knowledge and skills are interrelated with programming will be significant in improving method of instruction which is seriously underdeveloped at present. Knowing more thoroughly on how and what individual variables are affecting programming learning may let the educational administrators reconciling resources in providing a better programming learning environment for students.

Another valuable aspect of the study is its implication on the value of programming education. Investigating how and what componential knowledge and skills are related to problem solving in programming may provide important information for further researches on the transferrability of the problem solving skills polished through programming learning to other domains. This direction of research is meaningful only when programming itself does require such knowledge and skills. Findings from such researches will provide empirical data in response to the current assumption that programming education is valuable in promoting problem solving skills. If empirical results do not support the current claims, then the value of programming education should either be rethought or the context of programming education be restructured to meet the aim.

CHAPTER II REVIEW OF RELATED LITERATURE

This chapter will discuss literatures related to problem sovling in programming. Section 2.1 will discuss what knowledge components are related to programming. Section 2.2 will discuss what skills are involved in the problem solving process in programming. Section 2.3 will discuss the interrelationships between knowledge and skill involved in solving programming problems.

2.1 PROGRAMMING KNOWLEDGE

Weinberg's lankmark text, The Psychology of Computer Programming (1970), was the first book in providing insights on the psychological aspects of programming. Though lacking empirical evidence, Weinberg had pointed out from experiences that memory capacity is a facet of programming intelligence (P.167). This section will discuss models and empirical envidences on how and what programming knowledge is structured in the Long Term Memory (LTM) and how they are applied in programming composition.

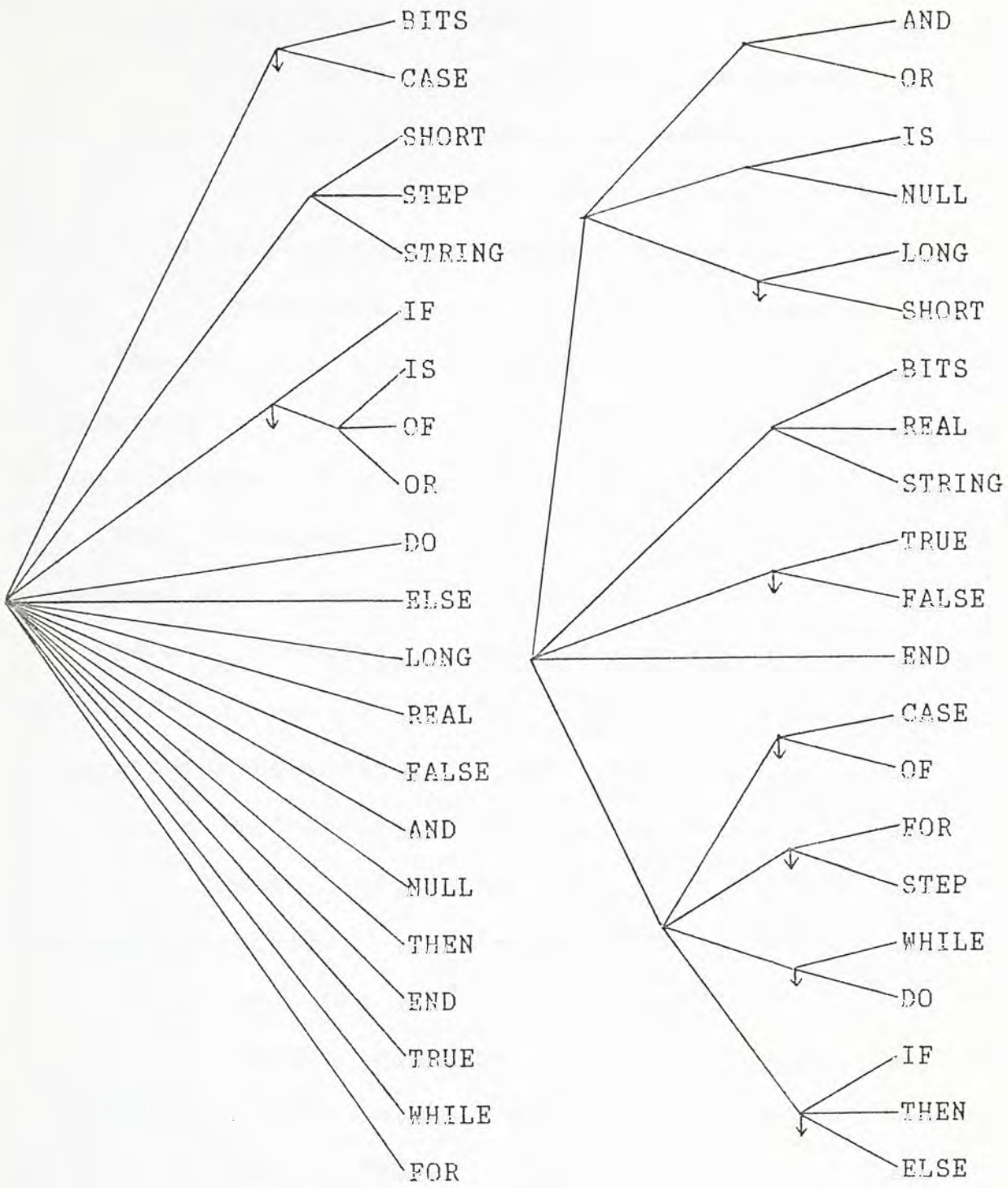
Two Types Of Knowledge In Computer Programming

The information processing theorists have pointed out that there are two structures in LTM which are essential in problem solving, namely, the knowledge structure and the heuristic structure. In a simple language of "process" and "data", problem solving involves retrieval data from the knowledge structure which are then operated by the strategies in the heuristic structure (Rumelhart & Norman, 1981).

Whether a problem can be solved successfully or not depends on both the sufficiency of knowledge in that domain as well as the presence of strategies applicable in that problem situation (Glaser, 1984). Research findings (Mayer, 1985; Jeffries, Turner, Polson & Atwood, 1981; Mckeithen, Reitman, Rueter, Hirtle, 1981; Adelson, 1981; Brooks, 1977; Shneiderman & Mayer, 1979) indicate that there are two types of knowledge in programming, namely, language knowledge and abstracted programming knowledge. Expert and novice programmers show differences in these knowledge organizations and access.

Programming Language

Writing program codes requires the ability to access efficiently one's knowledge on a programming language. Related studies by Adelson (1981) and McKeithen, et al. (1981) show that experts have a very highly organized base of knowledge about the programming languages. Experts show remarkably similar organizations based clearly on programming knowledge. On the contrary, beginning programmers' organizations show a rich variety of common language associations to these programming concepts. Figure 2 shows typical organizations for beginner and expert programmer in recalling ALGOL W reserved words.



A beginner´s organization apparently based on orthography

An expert´s organization apparently based on programming concepts

Figure 2 : A beginner´s and an expert´s organization of 21 ALGOL W reserved words (Mckeithen, et al., 1981)

Abstracted Programming Knowledge

In addition to language knowledge, programming requires a large amount of complex and detailed knowledge. Brooks (1977) analyzed protocols of programming sessions and concluded that programmers probably know tens of thousands of rules to represent all of a programmer's knowledge. Mckeithen et al. (1981) found out that expert computer programmers can recall at a glance far more information relevant to their field than novices can. Experts have not only more information, but also have them better organized into meaningful chunks. Bushke (1976) and other have shown that the more chunking one does, the better the recall.

Jeffries et al. (1981) showed that experts store well-integrated representations of previously solved algorithms. The experts demonstrated an impressive ability to retrieve and apply relevant information in the course of solving programming problems. The appropriate facts are utilitized just when they are needed and important items are seldom forgotten. On the contrary, novices lack adequate knowledge organization for solving programming problems (Pintrich, Berger, Stemmer, 1987). They frequently fail to apply correctly the knowledge which is needed to solve the problem, and the information which they require in the course of solving the problem is often not available to them when it is most needed. Jeffries et al. attribute this, in part, to the inadequacy of the organizing functions provided by their immature design schema.

Adelson (1981) studied differences between experts and novices using the PPL (Polymorphic Programming Language).

Subjects were shown three very short program, one line at a time. Altogether 16 lines were shown. Subjects were asked to free-recall as many items as possible. Experts recalled more than any other groups and had a different organization in the recall compared with the novices. These differences were explained in terms of syntactic and semantic encoding. Adelson concluded that (P.431)

> "both experts and novices have conceptual categories for the elements of a programming language. For the novices, the categories seem to be syntactic rather semantic in nature and the items within the category are not related to each other in a strongly organized manner. As expertise increases, the nature of the categories changes. They become more conceptually complex, changing from one line operations to entire routine, and they shift from being syntactically based to being semantically based."

Adelson (1984) supported the notion that novices encode programs more syntactically and experts more semantically by showing that novices performed better on tasks where performance benefited by syntactic encoding and vice versa for experts. Adelson suggested that the experts have a better level of abstraction on the programming knowledge and have them more hierarchically organized. Similar work were done by Shneiderman (1976b) and McKeithen et al. (1981).

The Syntactic/Semantic Model Of Programming Knowledge

In order to explain empirical results and provide a basis for making predictions, a cognitive model of programmer behavior based on experimental results (e.g. Shneiderman, 1976a; 1976b; 1977a; 1977b; Shneiderman, Mayer, McKay & Heller, 1977) was proposed by Shneiderman and Mayer (1979). The framework of the model is based on the information processing approach for problem solving. The cognitive model describes the cognitive structures and the cognitive processes involved in programming composition, testing, modification and learning.

Syntactic And Semantic Knowledge In LTM

A complex multi-leveled body of knowledge about programming concepts and techniques storing in the LTM are believed to be developed by experienced programmers. There are two parts of the knowledge, namely, semantic and syntactic. Semantic knowledge consists of general programming concepts that are independent of specific programming languages. Semantic knowledge includes low-level concepts of what an assignment statement does, what a subscripted array is, what data types are; intermediate notions such as interchanging the contents of two variables, summing up the contents of an array, a strategy for finding the largest of a set of numbers; and higher-level strategies such as binary searching, sorting and merging methods. All of this semantic knowledge is abstracted through experience and instruction in dealing with programming problems, but it is stored as general, meaningful sets of information that is independent of

the syntactic knowledge of particular programming languages.

Syntactic knowledge is a second kind of knowledge stored in LTM. It involves the understanding of the features or non-decomposable elements of the language such as details concerning valid character sets, names of library functions, arithmetic and relational conditionals, names of statements, or format of loops. Such technical details are more precise, detailed and arbitrary than semantic knowledge. Syntactic knowledge is specific to a particular language but semantic knowledge is generalizable over many different syntactic representations. Therefore, it is apparently easier for students to learn a new syntactic representation for an existing semantic construct than to acquire a completely new semantic structure. This is reflected in the observation that it is generally difficult to learn the first programming language but relatively easy to learn a second one.

Learning a first language for programming requires both specific details of the syntactic features of the language and the development of semantic concepts through experiences and abstraction in the process. The latter knowledge is independent of any particular language. Learning a second language with similar semantics involves learning a new syntax only. However, learning a second language with radically different underlying basic concepts in construction will be as hard or even harder than learning a first language. BASIC, and PASCAL are languages of similar semantics, but LISP is different in construction. Figure 3 serves to summarize the knowledge organization in LTM of an experienced programmer who has learnt various programming languages.

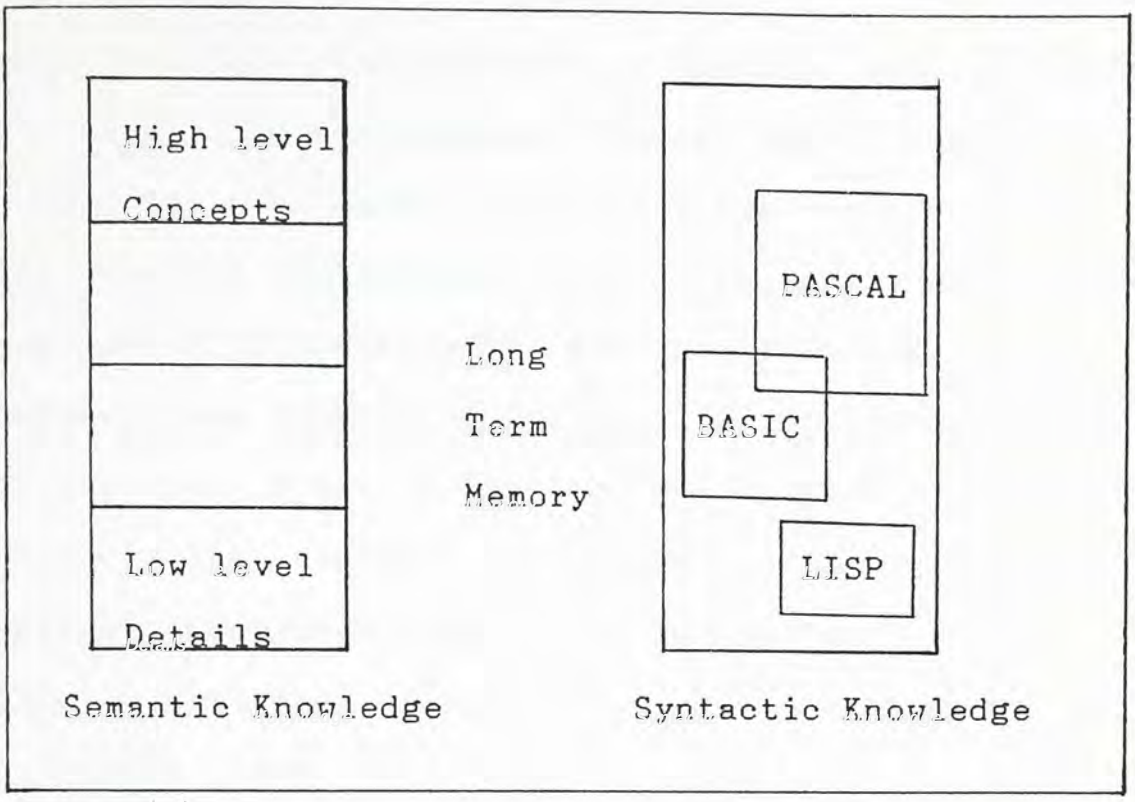


Figure 3 : LTM structure in the syntactic/semantic model

Shneiderman and Mayer (1979) further distinguish syntactic and semantic knowledge as two distinct structures in LTM from their means of acquiring. Syntactic knowledge, which is arbitrary and instructional, is acquired by rote memorization. The semantic knowledge, on the contrary, is acquired largely through intellectually demanding meaningful learning including problem solving and expository instruction. The former is more easily forgotten but the later can be retained more permanently. This distinction relect a basic difference between syntactic and semantic knowledge relevant to programming. In programming education, a compromise in instruction between syntactic knowledge and semantic knowledge which provides direction for creating strategies of solution mau be needed (Kreitzberg & Swanson, 1974).

Programming Plan

Following the research work of Adelson (1981), McKeithen et al. (1981) and Shneiderman & Mayer (1979) on demonstrating that experts have more and better organized knowledge than novices, Enrlich and Soloway shift in focus the empirical work on programming to identify the specific knowledge which expert programmers have (Enrlich & Soloway, 1984; Soloway & Ehrlich, 1984; Soloway, Bonar, & Ehrlich, 1983). The studies address the issue of the content of programming knowledge besides syntactical possessed by expert programmers. The basis approach is that they theorize that expert programmers encode their higher level knowledge in the form of plans which represent many of the stereotypic actions in a program.

There are two main kinds of plan, namely, CONTROL FLOW PLANS AND VARIABLE PLANS. CONTROL FLOW PLANS represent looping such as the running total plan in a program finding average of a set of numbers. VARIABLE PLANS represent various apsects surrounding the use of a variable such as the counter variable plan and running total varaible plan in counting and accumulating sums. Enrlich and Soloway theorize that programmers think about program design using these stereotypic plans which they have learned from experiences. Their notion of program plans is similar to the concept of schemata, which Norman, Gentner, and Stevens (1976) described as generalized representations of common algorithms.

In the experimental studies, novice and non-novice programmers are given a program in which lines of code have been replaced by blank lines. Their task is to fill in the blank lines with appropriate lines of code. The results of

the studies indicated that experienced programmers filled in the blanks correctly and in a plan-like fashion. Enrlich and Soloways' studies showed that programmers rely on their stereotypic plans when solving programming problems.

Anderson et al. (1984) use protocol analysis to study how students learn LISP. After carefully observing three students during their first thirty hours of using a LISP text, they report that students fail to learn from the text but rather use the templates, or structural analogies provided by examples in the text. However, when students apply these templates, they frequently fail to apply them correctly because of what they call "working memory failure". They mean the learner's inability to keep track of the conditions under which the template applies. They point out that the limited examples in the text are usually constraining students' learning for the structures.However, such template will become more robust as more programming knowledge is abstracted.

These studies indicate that programming requires organization of a large amount of complex and detailed knowledge about language syntax and abstracted templates. Templates are defined by Linn (1985) as stereotypic patterns of code which are employed as an entity in program to perform commonly encountered tasks. Learning syntactical language features and becoming familiar with many frequently used building blocks such as finding average of a set of numbers, sorting and searching routines are important factors in developing ability in solving programming problems. Thus coding appears to be a cognitive skill that can often become more quickly and easily as experiences are gained.

2.2 PROBLEM SOLVING PROCESS IN PROGRAMMING

Expert programmers not only have more available knowledge schema on programming, but also have strategies and rules which are applicable in solving problems. Programming design has been generally accepted as a problem solving process in the information processing approach (Brooks, 1977; Shneiderman, 1980; Pea & Kurland, 1987). This section will discuss related literatures treating programming as a problem solving process and relevant skills involved in it will be highlighted.

Computer Programming

Since computer programming occurs frequently in commercial and scientific situations, its task needs further clarification in this study. In many situations, programming includes a set of tasks like defining problem, writing program specifications and implementing them. On the cognitive processes in computer programming, Brooks focuses programming as a single task. In this task, a programmer is given a problem situation which may include the input data and the processing that is to be performed on it. The programmer finds out an algorithm, including the selection of internal representations for the data, and implement the algorithm in a programming language (Brooks, 1977). The theory further hypothesizes that programming has three distinct states, namely, problem understanding, method-finding and coding.

Dalbey and Linn in their literature research (1985) on the demands and requirements of computer programming concluded that programming is consisting of a number of identifiable

stages, each requiring a different kind of cognitive activity. Although none of the computer scientists and cognitive psychologists agree on the precise definition of the stages, in general, their analyses describe four tasks. They are problem specification, design, coding and debugging.

Considering programming in a learnable aspect in the pre-college setting, Pea and Kurland define the core sense of programming as the set of activities involved in developing a reusable product consisting of a series of written instructions that make a computer accomplish some task (Pea & Kurland, 1987). In terms of what a programmer does, a set of activities is involved in programming for either novices or experts, which constitutes phases of the problem solving process (Newell & Simon, 1972; Polya, 1957). These activities include four subtasks, namely, understanding the programming problem, designing and planning a programming solution, writing the programming code that implements the plan, and comprehension of the written program and program debugging (Pea & Kurland, 1987). Understanding the problem is a cognitive process that includes generating some representation of the problem which encompasses the recognition of the type of programming problem presented and activating the relevant schemas or templates to begin solution. Designing and planning the program includes specifying the overall flow of the program and selection of the procedures to be used. Coding the program includes writing the actual program code. Debugging involves the various strategies used to modify the program to have it operate properly.

A Theory Of Problem Solving In Programming

The following is an outline of a theory of processes involved in solving a programming problem proposed by Jeffries et al. (1981). The theory beleives that programming is a task involving the coordination of a complex set of processes. It includes applying abstract knowledge about the problem representation and involving the retrieval of computer knowledge or information from LTM about the problem. It also involves the storage of relevant information in the short term memory (STM) for later use in solving problems. Most fundamentally, the theory focuses on the design task, particularly its guiding control processes, and on the manipulation of knowledge within the problem solving effort.

Design Schema

Experts have knowledge concerning the overall structure of a good design and of the process of generating one. This implies that skilled programmers have knowledge describing the structure of a design independent of its content. This abstract knowledge about design and design processes, along with the set of procedures that implement these processes, will be referred to as the design schema.

This schema will develop through experience with programming. Originally, a programmer's approach to a problem is assumed to involve general problem solving strategies, such as "divide and conquer". As an individual has more and more experience with this activity, these general strategies are transformed into a specialized schema. The schema is developed through the addition of domain-specific concepts,

tacits, and evaluative criteria. Whenever a solver's specialized schema is inadequate to solve a problem, more general strategies take over.

It is proposed that problem solving in programming is the interaction between the design schema and the more specific knowledge involved in the problem. The design schema is not tied to any specific problem domain but consists instead of abstract knowledge about the structure of a completed design and the processes involved in the generation of that design.

There are four components in a design schema. First, it contains a collection of components that partition the given problem into a set of meaningful tasks. Second, it contains a collection of components that add elements to tasks in order to assure that they will function properly, for example, the initialization of data structures or loops in a program. Third, it contains a set of processes that control the generation of designs. Finally, it contains a set of evaluation and generation procedures that ensure effective utilization of knowledge.

Each component of the design schema is composed of both declarative and procedural knowledge about the abstract nature of the design process. Declarative knowledge refers to computer knowledge such as language features and templates. Procedural knowledge refers to strategies such as planning, decomposing problems into parts and analogical reasoning. The design schema can be applied recursively that leads to a modular decomposition of the problem into more and more detailed modules.

The major control processes of the design schema are

summarized as a set of very abstract production rules. The rules mainly composed to two major categories. One concerns solution derivation process. The other concerns solution retrieval process. When no solution model exists in the solver's schema, a search strategy will be applied. When a solution model or a potential solution exists, the solution model or modified solution model will be applied to the existing problem or subproblem.

Problem Solving Strategies

Design schema in the theory of problem solving in programming proposed by Jeffries and his colleagues focus on the design stage of programming. In the theory, programming design is treated as the interaction of programming knowledge and design skills. Such design skills are abstracted through experiences and usually is tacit. In the information processing approach's terminology, such skills are called problem solving strategies. In general, a problem solving strategy is a technique that may not guarantee solution, but may serve as a guide in the problem solving process.

The information processing theorists in the seventy's mainly focused on investigating strategies in problem solving which can be applied generally. Early models of problem solving, such as the General Problem Solver (GPS) of Newell and Simon (1972), focused heavily on general search strategies through states in a problem space. Information processing psychologists believe that there are always the presence of general heuristics which help the solver to solve problems. Nevertheless, so far, theorists are far from arriving at a

commonly agreed set of general heuristics. The following strategies are some commonly addressed by many of the information processing theorists which are related to programming design. They include problem decomposition and analogy.

Problem Decomposition In Programming Design

Problem decomposition is breaking problem into subproblems. It is a useful general strategy in solving problems of design, ranging from the design in architectural problems (Simon, 1973) to the design of software (Jeffries, Turner, Polson & Atwood, 1981). The task of design involves a complex set of processes. Starting from a global statement of a program, a solver must develop a precise plan for a solution that will be realized in some concrete way. Very often, design tasks are too complex to be solved directly. Thus, an important facet of designing is decomposing a problem into more manageable subunits.

There are several decomposition strategies that a designer can use to guide the process in programming. One strategy is to break the problem into input, process, and output elements. The others are problem solving by analogy and problem solving by understanding (Jeffries et al., 1981).

Among the decomposition strategies, the input-process-output strategy is preeminently used. The initial pass at decomposition results in a representation of the problem. A model is devised specifying a set of tasks that will solve the problem and a control structure for these tasks. It is then expanded into a set of well-defined subproblems. The

solutions to these subproblems represent a solution of the original design problem. This process of decomposition is then applied to each subproblem in turn, resulting in more and more details plans of what should be done to accomplish the task.

Problem decomposition by analogy occurs when the solution model generated for a given subproblem, or some part of it, is reorganized as being analogous to an already understood algorithm. This attempt to retrieve previous solutions is invoked once a solution model has been derived, but before any further refinement takes place. The discovery of analogs for a particular problem may be influenced by the organization of programming knowledge (Pennington, 1982). The way programming knowledge in LTM is structured into meaningful chunks make it an appropriate domain for analogical problem solving. Analogical structure mappings between two problems could occur at the level of any of these units of knowledge. Since these knowledge units represent context independent problem solving structures, they are particularly suited to analogical transfer across problems (Clements, et al., 1986).

Problem decomposition by understanding is prominent in cases where an element identified by application of the design schema is not understood with enough detail for the design schema to be applied to it. The solver's knowledge of the problem area in question, as well as of computer science, is then used to refine the understanding of this element.

The Role Of Memory In Program Composition

A model of program composition is suggested by

Shneiderman and Mayer (1979). The model describes programming as a problem solving process in four stages proposed by Polya (1957), namely, understanding the problem, devising a plan, carrying out the plan, and checking the result.

Understanding a problem is a stage defining what is given and what is the goal. When a programmer receives a problem, it is assumed that it will arrive the working memory through the STM after effort is paid. The problem will then be analysed and represented in terms of the given state and goal state in the working memory. Syntactic and semantic knowledge and other general information will be called and transferred to working memory from LTM for further analysis in understanding the problem.

Devising a plan is a stage in which a general strategy of solution is generated. The model proposes that the problem solution is conceived of in general programming strategies and application-related domains. The plan is developed from a very general, to a more specific plan, to a specific generation of code focusing on minute details. The general plan is referred as internal semantics in the model.

There are usually three general approaches to plan a programming solution, namely, top-down, bottom-up, and modularization. A top-down implementation of the internal semantics for a problem would demand the highest levels be set first, followed by more detailed analysis. This process is a general problem solving heuristic called "working backward" by Polya. Top-down design is an approach which decomposes a complex problem into subproblems. Experts use their knowledge of templates to guide the decomposition process. Top-down

design is somewhat iterative in nature. Stepwise refinement (Wirth, 1971) is another top-down design technique in which the programmers are engaged in successive restatements of the problem specification with each step closer to machine level notation. Programmers applying this technique need to be very familiar with the language the machine uses.

A bottom-up implementation would permit low-level code to be generated first, in an attempt to build up to the goal. This process is referred to as "working forward" by Polya. Modularization is another technique in develop programming solution. Such an approach develop program modules to perform specific functions which are served as building blocks for the entire solution of the problem. Such a method of approach is termed as structured programming advocated by the computer scientists Dahl, Dijkstra and Hoare (1973). Polya and Wickelgren refer to this technique as making "subgoals" which is also a general problem solving strategy. Each of these techniques leads to the internal semantics of the programming solution.

Carrying out the plan is a stage in which the plan is translated into a specific course of action. Once the internal semantics have been worked out in the mind of the programmer, the code can be written out by using the syntactic and semantic knowledge in the LTM. The program may be composed easily in any programming language with which the programmer is familiar.

Checking the result is a stage in which the solution is tested to make sure it works. This is done by a computer run of the program.

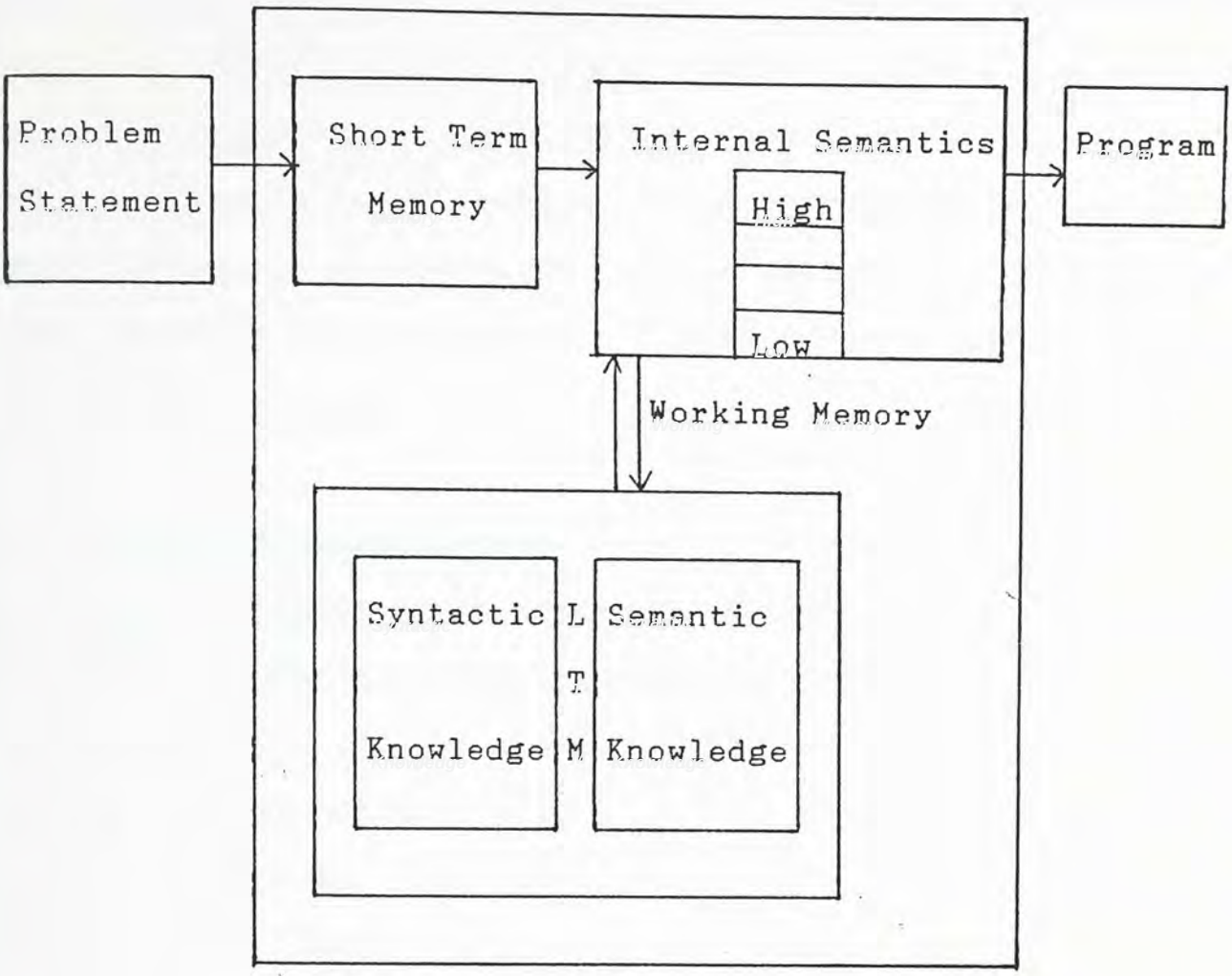


Figure 4 : Program composition process

When errors are detected under tesing, debugging and program modification are necessary. Debugging is a complex task to locate an error in a program. Syntactical errors can be resolved quite easily by referring to programming manual. Further types of bugs are either due to incorrect transformation from the internal semantics to the program statements or due to incorrect transformation from the problem solution to the internal semantics. The former type of error is usually caused by improper understanding of the function of certain syntactic constructs in the programming language or simply by mistakes. The later type of error is usually due to incorrect or improper program design such as failure to deal

with out-of-range error, inability to deal with special cases such as the sorting of a single value. Pinpointing the type of error detected, a modification task is required which must be reflected in an alteration of the internal semantics. The modification task requires skills gained in composition, comprehension and debugging. Figure 4 summarizes the program composition process.

The Problem Solving Process

Recent information processing theories of problem solving emphasize two important processes. One is the generation of a problem representation or problem space, that is the problem solver's view of the problem. The other is the solution process that involves a search through the problem space (Gick, 1986).

Problem Representation

When a solver encounters a problem situation, the solver will try to extract the given and goal information and attempt to understand the problem. Very often, in constructing a representation of the problem, the solver will connect the problem situation with existing knowledge in LTM so that an integrated representation can be formed in working memory. Processes involved in understanding include analysis of relations among elements and identification of patterns among the relations.

Schema Activation

During the construction of a problem representation,

certain features of the problem may activate knowledge in memory. For example, the presence of a certain key words and phrases in an algebra word problem may activate a schema in handling that type of algebra word problem (Kinstch & Greeno, 1985). Schema is a cluster of knowledge in the memory. Schemata are induced from past experience with numerous exemplars of the concept it represents. A schema can guide the organization of incoming information into clusters of knowledge. When knowledge is represented in terms of a schema, associations from the schema will faciliate retrieval in a later stage (Thorndyke & Hayes-Roth, 1979). When a schema related to a problem type exists, it will facililate the solver to use it in the solving process. Usually, a schema related to a problem type contains information about the typical problem goal, contraints and solution procedures useful for that type of problem (Gick & Holyoak, 1983).

When schema activation occurs during the phase of problem representation of problem solving, Gick describes the problem solving process as schema-driven. Little search for solution procedures is needed because appropriate solution procedures are activated by recognizing the particular problem type.

Search For A Solution

In the absence of appropriate schema activation, the problem solving process proceeds to the second stage and a search strategy is invoked. General strategies may involve the comparison of problem states to the goal state, as in means-ends analysis. Problem decomposition is also a general strategy often applied in breaking the problem into

subproblems and finding a suitable order of completion of the subproblems. A general search strategy advocated by Polya (1957) is problem solving by analogy, in which the solver looks for an analogous problem for which the solution is known. The details of these general strategies have been fully discussed in the previous sections.

Solution Implementation

No matter a schema-driven strategy or a general strategy is searched, the solver must proceed to the third stage of the problem solving process, that is, the implementation of solution strategies and procedures. If the solution is successful, the problem will be solved. If the solution is unsuccessful, the problem solving process will be reiterated by reformulating the programming solution depending on the nature of the failure. In programming problems, very often the failure or success is validated by a computer run.

The Problem Solving Process In Computer Programming

The above related literatures revealed that there are four subtasks of programming, namely, problem understanding, planning, coding, testing and reformulating. Integrating the above cognitive psychologists' point of views (Brooks, Jeffries et al., Shneiderman & Mayer, Dalbey & Linn, Gick), the following problem solving process in computer programming is summarized.

When a programmer receives a problem, the first stage is the construction of a problem representation. Following the representation, the programmer may activate a schema for that

particular type of problem from LTM and implement it by analogical reasoning. If no schema is activated, the programmer will then search for a solution to the problem by using more general heuristics like problem decomposition. The implementation process includes coding, and testing. If the solution is successful, the task is over. It will backtract to an earlier stage if the solution fails. The solver may attempt to reformulate the problem or use another method to solve it, depending on the nature of the failure. Figure 5 serves to illustrate the process.

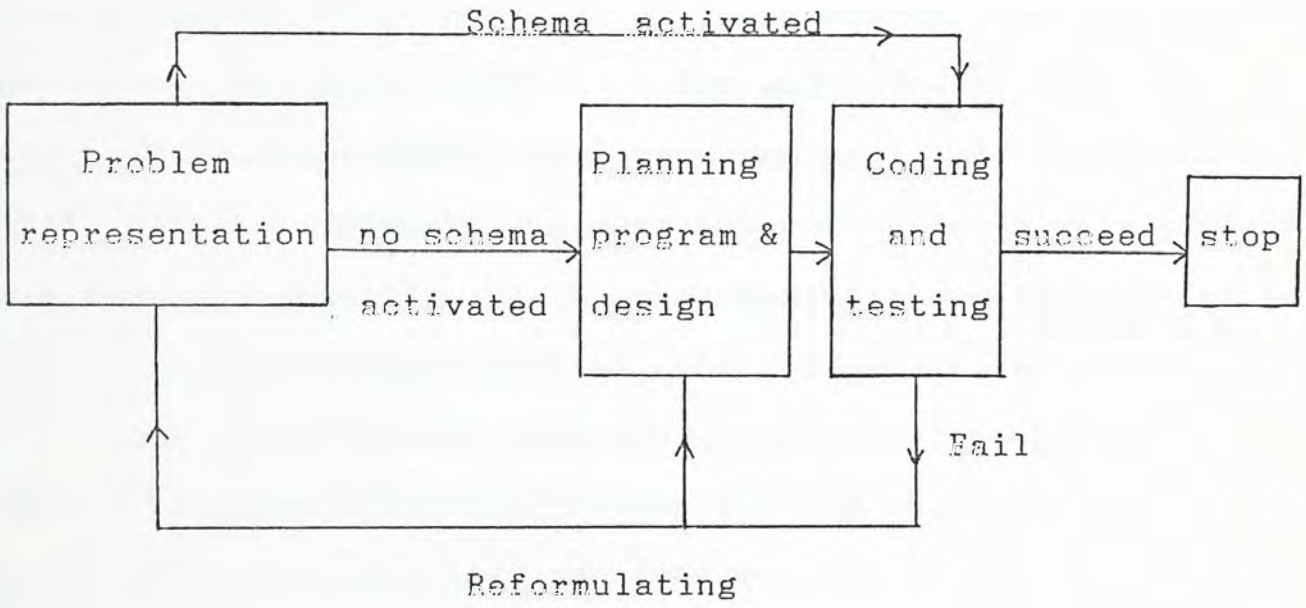


Figure 5 : Schematic diagram of the programming process

2.3 THE ROLE OF KNOWLEDGE AND SKILLS IN PROGRAMMING

Section 2.1 presented that language syntax and abstracted template are knowledge components needed in programming. Section 2.2 revealed that procedural skills including planning, testing and reformulating are indispensible elements in the programming solving process. This section will discuss related literatures depicting the role of knowledge and skills in programming.

Cognitive Consequences Of Programming Instruction

A model called the chain of cognitive consequences for understanding the learning of programming instruction is presented by Linn (1985). The model describes a chain of cognitive accomplishment for programming. It hypothesizes that computer programming does not only promote knowledge of the features specific to a programming language but also skills in planning, testing and reformulation. The chain includes three links, namely, language features, design skills, and generalizable problem solving skills. The present study focuses on the language features and design skills only.

The Chain Of Cognitive Accomplishments

The first link in the chain of cognitive accomplishment consists of language feature, the element which is basic to all formal systems. They are the non-decomposable features of a programming language, for example, INPUT, PRINT, IF...THEN, FOR, NEXT or GOTO in BASIC.

The second link in the chain consists of design skills. Design skills are group of techniques used to combine language

features to form a program that solves a problem. Design skills include templates and procedural skills. In programming, templates are stereotypic patterns of code that combine several language features to accomplish a specific function. Common programming templates are summing an array, sorting a list of integers or searching an element in an array. Templates reflect how learners organize programming knowledge. Templates perform a function similar to schemata in the theoretical formulation of Anderson (1984) and to plans in the work of Soloway and Ehrlich (1984).

Procedural skills are used to combine templates and language features in order to solve new problems. Procedural skills include planning, testing, and reformulating. A plan is used for combinimg language features and templates to solve a programming problem. Planning includes decomposing the problem into component parts and combining those parts. Once a plan is designed, then it will be implemented by coding into a program in a particular computer language. The program will be subject to test in order to ascertain its correctness. Testing involves determining whether a program meets specification as required in the problem. Test data will be designed and implemented to see whether the program can operate correctly. When testing of a program reveals problems, it will be decided whether it requires refinement. Reformulating is a technique for modifying programs to make them work properly.

The third link on the chain consists of problem solving skills useful for learning new formal systems. These are templates and procedural skills common to other formal systems

including learning new programming languages. These general problem solving skills may be acquired when students attempt to apply templates or procedural skills learned in one system to a new one. Students probably need experience with several formal systems before they acquire general problem solving skills.

Language Features, Design Skills And Programming

The present study is not intended to explore the transferability of the problem solving skills induced from programming. It is intended to investigate the prerequisite knowledge and skills for proficient programming. According to Linn's model, proficient problem solving in programming requires both language features and design skills which should be acquired in the order as illustrated in Figure 6.

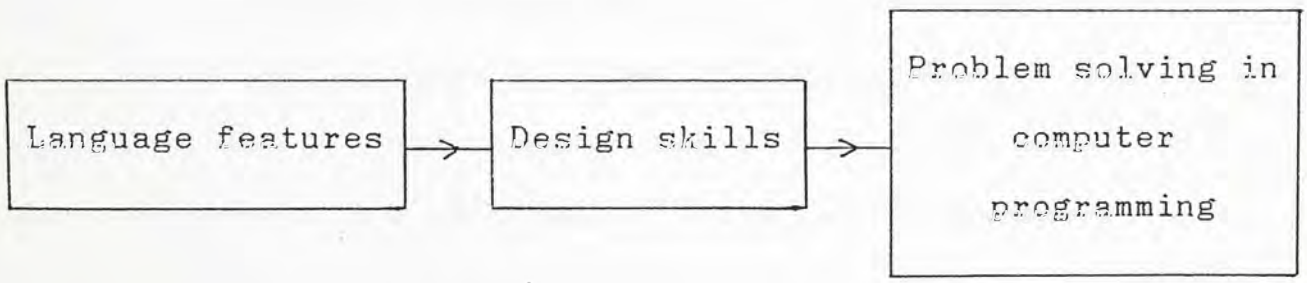


Figure 6 : Schematic diagram for Linn's model on programming

The assumption that earlier components are prerequisite for later components in the chain is an excellent candidate for empirical testing (Webb & Shavelson, 1985). The following sections will discuss with technical details on how the model can be tested empirically.

Language Features In Programming

Language feature is often the sole topic of texts and programming courses in pre-college settings. Instruction in language features gives definitions, examples of correct statements and have students use them in computer sessions. This knowledge pertains on how to recognize and interpret valid statements in the language, and how to construct correct statements. This is a rather domain specific and detailed knowledge.

Sophisticated programming activities demand high level thinking skills and problem solving techniques. These activities require expertise. They require a good model of the structure of the language. A programmer cannot write a modular program and use variables unless he understand how control is passed in the language and how logical tests operate on outputs of operations (Kurland et al., 1986). Thus, the problem solving ability in programming depends upon students' proficiency in the language features. However, knowledge on language is not a sufficient condition for solving programming problems. There may exist some mediating factors that influence ability in solving programming problems.

Assessing Understanding On Language Features

Students' knowledge of language features is often assessed by comprehension of already coded programs, and by reformulation of programs (Dalbey & Linn, 1986). Comprehension and reformulation of language features are always the primary activities in most programming courses.

Language feature knowledge is often assessed by comprehension items asking students to predict outputs from programs using the features. It is also assessed by asking students to change a language feature in a program so that the program does something slightly different. This may include change print messages in programs, modify arithmetic expressions, alter the boundaries of a loop, and other simple changes to already written programs.

Design Skills In Programming

Students with understanding of language features may compose very simple programs. However, it cannot work when the task become a little bit complicated. Boulay (1986) shares the gestalt's point of view in distinguishing language features and program understanding in the following way :

> "Even after becoming familiar with some of the pieces that make up a program, e.g., PRINT or IF, the novice may not see a program as a working mechanism in the same way as an experienced programmer. This ability to see a program as a whole, understand its main parts and their relation is a skill which grows out gradually."

Therefore, students need to learn language features, but such knowledge is not sufficient for composing programs or even understanding programs. Linn proposes that design skills are needed to invent or refine programs (Dalbey & Linn, 1986). These include templates and procedural skills. Such knowledge

and skills are essential in order for students to write computer programs of any complexity. Review of programming researches indicated that design is the most important cognitive activity in programming (Dalbey & Linn, 1985).

Templates

Templates are chunks of code longer than a single line that perform reasonably complex functions (Dalbey & Linn, 1986). As a student learns about programming, he may start to recognize typical algorithms or recurrent elements in the solutions to similar problems. These stereotypic patterns are referred as templates. Templates are abstractions at a higher level than specific language features. Hence it must be acquired after language features has been grasped. Templates reflect how learners organize programming knowledge and they perform a function similar to schemata as has been discussed in the previous sections.

Templates can be represented as code, pseudo-code, or material language descriptions in books. Books of algorithms provide examples of templates (Linn, Sloane, and Clancy, 1987). Anderson and his colleagues (1984) find out that students do learn templates from programming examples in texts. Students learn templates from examples in class or in textbooks and they generalize templates when they discover new uses for the templates or notice cases that it does not cover.

Templates can be complex in that they depend on other templates, or self-contained in that they end the action and return attention to the calling template. Templates can be general to most programs or specific to programs with certain

features. Templates exist at a variety of levels. The lowest level templates are language statements use in one particular situation. Higher level templates are general enough to be applicable in a variety of situations (Linn, et al., 1987).

Templates may help a solver to solve programming problems. When students have a repertoire of templates, they have a set of flexible and powerful techniques which allow them to solve many programming problems without inventing new code (Linn & Dalbey, 1985). Furthermore, templates help a solver to plan in designing program (Soloway & Ehrlich, 1984). Templates enable the programmer to design solutions in a more top-down manner because templates reduce the cognitive demands of programming. Programmers who have a repertoire of templates may simplify complicated procedures by recalling reusable chunks of code. In other words, templates provide obvious ways to decompose a problem. Programmers can decompose problems into pieces according to the size of their available templates. Then they can implement the solution by using their templates. Students who have a repertoire of templates, therefore, can write more complicated programs than those without such a repertoire.

Templates may also help testing programs. If programs are written in segments, then testing each template to isolate errors becomes more systematic (Mandinach & Linn, 1986). Reseachers have also begun to investigate the role of templates in debugging (Weiser, 1982). Johnson and Soloway have produced a knowledge-bases system which uses information about templates and transformations on them to diagnose semantic bugs in freshman programs (Johnson & Soloway, 1983).

Assessing Repertoire Of Templates

Nutter and Hassell call templates as schemes. They define scheme as an abstract structure which captures the essential features of a process to solve some problems. These schemes contain the conceptual, as opposed to machine-dependent, information of what a particular chunk of code does (Nutter & Hassell, 1985). It is therefore important to distinguish between these underlying conceptual patterns and their embodiments in pseudo-code, program code or structure chart, or any other form. A particular instantiation of a programming scheme in some concrete form is called a scheme representation.

Shneiderman & Mayer (1979) have also suggested that programmers develop an internal semantic structure to represent the syntax of the program. They do not memorize or comprehend the program in a line-by-line form based on the syntax. When asked to recall the templates, they applied their knowledge of a language syntax to convert their internal semantics back into a program of a particular language.

```
REPRESENTATION  A
COUNT <--- 1
While not End of Input
  read ARRAY [COUNT]
  COUNT <--- COUNT + 1
end while
COUNT <--- COUNT - 1
```

```
REPRESENTATION  B
COUNT <---0
While not End of Input
  COUNT <--- COUNT + 1
  read ARRAY [COUNT]
end while
```

Figure 7 : Scheme representations in reading an array

In order to assess students' repertoire of templates, we can actually only assess their representation in a form, like in program code. Nutter & Hassell (1985) have given examples on scheme representations as shown in Figure 7.

Templates exist at a variety of levels. The lowest-level templates are language statements to use in one particular situation such as comparing two numbers; knowing what an assignment statement does, and what data types are. Intermediate-level templates are small algorithms such as summing an array, finding the mean of a set of numbers, interchanging the contents of two variables, and finding the least of a set of numbers. Higher-level templates are larger algorithms such as sorting, searching,finding the least common multiple of a group of numbers, counting the number of words in a text, and even larger units such as an entire program pattern in an application area (Shneiderman & Mayer, 1979; Linn, Sloane & Clancy, 1987; Linn, 1985; Linn & Dalbey, 1985; Enrlich & Soloway, 1984). The examples mentioned above are the possible knowledge and form of representations that may be applicable in assessing students' repertoire of templates.

Procedural skill

After learning specific features of a programming language, students have to learn how to create an entire program. Students must plan how commands are combined to reach a desired goal. The resulting program must then be tested for correctness and errors must be isolated. Finally the errors must be eliminated by reformulating the program.

Therefore procedural skills include planning, testing and reformulating. These skills are necessary for logical and efficient design of computer programs.

Planning

Planning is believed to be useful to solve complex programming problems. However, there are two approaches toward planning in the problem solving literatures on planning strategies. One approach involves preplanning activity, the other does not. Hayes-Roth and Hayes-Roth (1979) are the former advocators. They call preplanning as top-down or hierarchical planning which involves developing a plan for solving a problem before beginning to solve it. The later approach is advocated by Pea, Kurland, Papert, Watt, diSessa and Weir. They term planning in action as opportunistic planning or bottom-up programming, problem-solvers use current decisions and observations to make decisions about what to do next. Opportunistic planning involves incremental steps and may not lead to a coherent, integrated plan (Webb, Ender & Lewis, 1986).

It is believed that planning will facilitate solving complex programming problems. Linn believes that programming problems which involve only linear combinations of simple language features often fail to illustrate the advantage of planning. Strategy in decomposing problems into subproblems, and plan how to combine those parts is believed to be beneficiary to complex programming. Programmers use this decomposing strategy to develop steps needed to solve a problem. One strategy is to identify parts of the problem

that can be performed with known templates. Another strategy is to identify similar actions and group them together so that they can be performed with a single subroutine. When difficult parts are encountered in designing, a more general strategy like to separate the task into input, process, output subtasks will be employed. Expert programmers may also intersperse designing with implementating. Easy parts of the program may be implemented first and then return to design the difficult parts so as to reduce the complexity of the task (Linn, Sloane, & Clancy, 1987). Other strategies like subgoal generation, modular documentation, retreival of known solution are also believed to be useful in program planning by rational analyses and obervations of adult programmers (Pea & Kurland, 1984).

Assessing Planning Skill

Though researches indicate that expert computer programmers spend much of their programming time in planning the solution before beginning to write computer code (Atwood, Jeffries, & Polson, 1980; Kurland, Mawby, & Cahir, 1984), on the contrary, novice computer programmers show little advance planning (Dalbey, Tourniaire & Linn, 1986; Galanter, 1983; Webb, Ender & Lewis, 1986). The influence of planning on problem solving in programming in pre-college settings is still not clear. Linn has pointed out that problems which involve only linear combinations of simple language features often fail to illustrate the advantage of planning (Linn & Dalbey, 1985).

Acquisition of program planning skill can be assessed by

asking students to write programs to solve problems. To assess planning skill, such problems must be reasonably complex (Linn, 1985). In addition, planning with a sense of available templates in mind will reduce the complexity of the task because some of the smaller program segments will already be written. In order to assess whether larger repertoire of templates will facilitate planning, programs must require use of commonly learned templates. Assessing planning skill by writing reasonably complicated program cannot distinguish preplanning or planning-in-action behavior. To survey students' planning behavior, questionnaire on planning activity for each programming problem can be conducted.

Testing

Testing a program is to find out whether a program perform as intended. Students always fail in applying the testing technique to check whether their answer is correct in mathematics problem solving (Polya, 1957). In programming, experts and novices differ in this skill. Experts not only recognize the advantages of testing their programs but are also good at designing tests to reveal possible problems. They have well-developed knowledge of antibugging strategies associated with their templates. Experienced programmers have strategies for enumerating the conditions under which the program should perform. These may include such conditions as testing for bad input, testing for lack input, testing for input in the wrong order, or for other potential problems. In contrast, novices often test only the obvious or usual forms of input and may fail to test all of the code (Mandinach &

Linn, 1986; Linn, Sloane, & Clancy, 1987).

Specific testing strategies are associated with particular templates. Programmers generally test loops to see if they are off by one. They test arrays to determine whether they handle the end elements correctly.They test sort routines to see if they can sort two elements. Programmers organize their knowledge so that templates point to testing strategies.

Assessing Testing Skill

In order to assess students' testing skill, students may be asked to suggest test data for coded programs and implement additional build-in-tests in given programs. As Linn has pointed out that experts tend to test the boundary conditions, to insure that no division by zero is possible and to consider difficulties resulting from interactions between parts of their program. In addition, programs written by experts tend to have build-in tests for potential confusions such as tests to be sure that the input data meet the problem specifications. Therefore, asking for test data and build-in tests in given program to meet problem specifications will be sufficient in assessing students' testing skill.

Reformulating

When errors are found in testing, programmers reformulate programs to make them work properly. It is another skill that differentiates experts and novices. Experts are likely to respond to the results of tests by considering large-scale as well as minor reformulations of their programs. In contrast, novices tend to seek localizes "fixes" for their

programs (Atwood & Ramsey, 1978), perhaps never learning how to revise larger programs. Reformulating program plans, both major and minor changes, depend on ways in finding out the sources of errors and ways to eliminate them. Finding out errors are also called debugging in programming. Debugging strategies specify actions for identifying the bug that generated the symptom revealed by the test. As mentioned earlier, debugging activities ranged from debugging low level syntax errors to high level logical errors in planning (Shniederman & Mayer, 1979).

Debugging stategies involving program design can also be specific to a particular template (Linn, Sloane, & Clancy, 1987). When one element is missing from a sort, expert programmers often suspect that the first or last element is deleted. When output is way off, programmers may look for loops that fail to process one element. Programmers associate particular symptoms with certain bugs.

Assessing Reformulating Skill

Reformulating skill can be assessed by asking students' to modify given programs which does not meet the required specification. Program need modification should contain templates for reformulating.

Assessing Problem Solving Ability In Programming

Problem solving ability in programming can be assessed by asking students to write programs to solve problems. Such problems must be reasonably complex so as to involve skills on planning, testing and major reformulations. Problems must

also require commomnly learned templates in order to see whether students with larger repertoire of templates will have better performance in programming.

Language Features, Templates And Procedural Skills

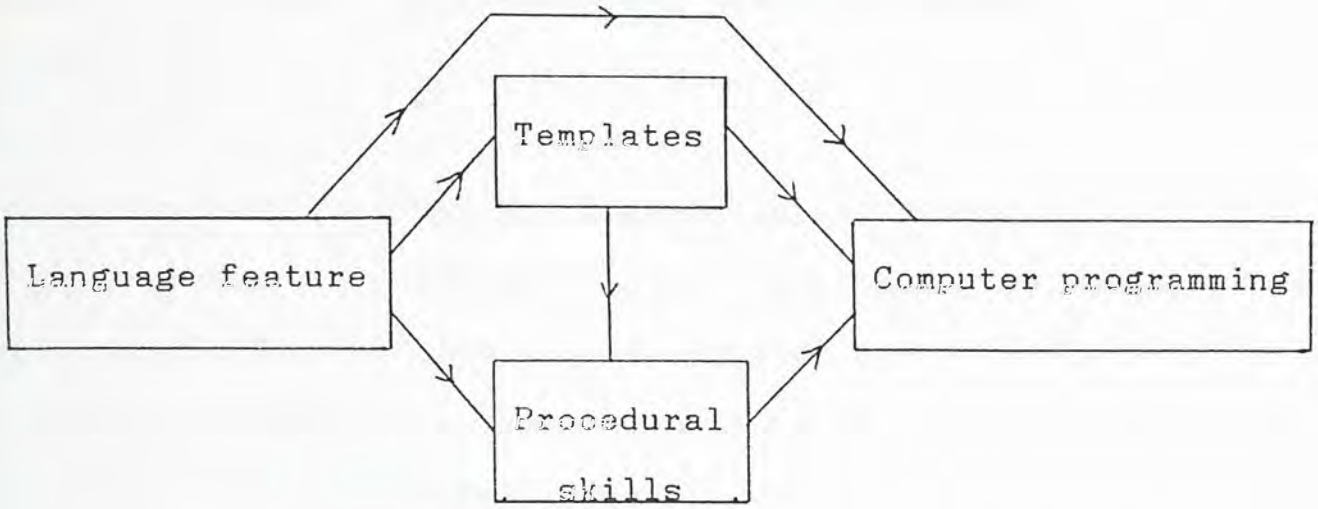


Figure 8 : Theoretical path model in learning programming

Let's summarize the role of language features, templates and procedural skills in programming. Linn has postulated that language features is the first link in learning programming. Such knowledge is necessary in using the language but is not sufficient to foster higher cognitive skills such as design skills in programming (Mandinach, Linn, Pea & Kurland, 1987). Design requires templates and procedural skills. As students come across a variety of programming examples, exercises and become proficient in a programming language, templates and procedural skills must be abstracted either tacitly or explicitly through instruction. These abstracted programming knowledge and skills will foster programming ability either directly or indirectly. As it has

been thoroughly discussed in the previous sections that template will facilitate planning, testing and reformulating skills in programming which will in turn foster programming ability. Figure 8 summarizes the role of language features, templates and procedural skills in computer programming. This theoretical path model will be tested empirically in this study.

Individual Differences On Programming Performances

Factors hypothesized to influence outcomes from programming instruction include general abilities (high/low), gender differences, home computer possession, and teacher-student ratio (high/low).

General ability measured by Raven's Advanced Progressive matrices (Raven, 1958) was found more significant in learning language features, but not so significant in learning design skills and programming (Linn & Dalbey, 1985; Fisher & Mandinach, 1985). Correlations of ability and success in programming in general ranged between 0.15 to 0.54. Dalbey and Linn account the volatility nature of these correlations to that programming requires a more precise set of skills than those measured by tests of general reasoning ability.

The categorization of computer in the society as a domain highly related to mathematics and science has laid down a tacit implication on sex-related differences (Hawkin, 1987). Research findings do indicate that boys are performaning consistently better than girls in knowledge of language features, debugging and program composition (Pea, Hawkins, Clement, & Mawby, 1984). It is hypothesized that these

differences are not due to intelligence but to social factors (Dalbey & Linn, 1985; Becker & Sterling, 1987).

Possessing home computer is found to be related to students' performances on programming in typical schools as they are involved more in language features. However, home computer possession is found to be not related to students' performances on programming in exemplary schools as they are involved more in designing programs (Linn & Dalbey, 1985; Fisher & Mandinach, 1985). It is accounted that the in-class computer access time in typical schools are limited. Therefore, out of school access reflects benefits from opportunities for drill on language features. At exemplary schools, sufficient computer time are provided in school for students to learn the design skills taught. Thus, it is postulated that when access is either extensive or efficient and when students are far enough along the chain of cognitive accomplishments that their concern is with design of programs, then the influence of out-of-school access is small.

Raising the teacher-student ratio in computer lesson is a new measure adopted in Hong Kong. It is worthy to study whether such a change will affect students' performances on programmming.

Summary Comments

In the literature reviewing process, the researcher observed that their are large volume of researches investigating the transferrability of problem solving skills polished from programming instruction to other domains. The results are not confirmatory. A research direction on

studying the contents in programming courses and instructional methods is called for. There are few researches studying in this aspect. The present research tries to make some contribution in this arena.

In the past, programming education was limited to professional programmers and college students. The target population for studying was limited to this small volume of people. Therefore, the researches done by Adelson, McKeithen, Brooks, Jeffries, Shneiderman, Mayer, Soloway and Enrlich were either on programmers or college students. As programming education are expanding to pre-college settings, the target population for research on programming should also be shifted. The present research will study secondary school students.

There are increasing number of researches studying the programming behavior of pre-college students. However, Johanson (1988) commented that the treatment durations on programming instruction were insufficient in most of these researches. Twenty to forty hours of programming instruction was typical. Webb's (1985) exploratory study involved only three hours of instruction. The present study will investigate students who have learnt more than 80 hours of programming.

Studies investigating template (e.g. Enrlich & Soloway, 1984), and planning in programming (e.g. Webb, Ender, & Lewis, 1986; Dalbey, Tourniaire & Linn, 1986), had been done in different researches separately. Such results cannot describe the role of template and planning in programming comprehensively. The present study tries to study these factors aggregately in programming.

CHPATER III METHOD AND PROCEDURE

3.1 Definitions

1. Programming

Programming refers to the set of activities involved in developing a resuable product consisting of a series of written instructions in a programming language that make a computer accomplish some task that is specified in a problem statement (Pea & Kurland, 1987).

2. Language features

These are the "primitive" or nondecomposable elements of the programming language. For BASIC they include "IF...THEN", "GOTO", "PRINT", and others (Linn, 1985).

3. Templates

Templates are stereotypic patterns of code using more than a single language feature. They are employed as an entity in programs to perform commonly encountered tasks (Linn, 1985).

4. Procedural skills

Procedural skills are used to combine templates or language features to solve programming problems. They include planning a solution using available templates and language features, testing the plan to see if it accomplishes the objectives and reformulating the plan until it succeeds (Linn, 1985).

3.2 Research Method And Hypotheses

This research measures various aspects of programming knowledge and skills separately and investigates their correlations with programming ability. The knowledge measured are language features and templates. Procedural skills measured are planning, testing and reformulating as defined above.

Language features and templates were measured by the Language Proficiency Test (LPT) and Template Measurement Test (TMT) respectively. Testing and reformulating skills were measured by the Test on Procedural Skills (TPS). Planning skill was measured by the Test on Problem Solving in Programming (TPSP). Scores on Procedural Skills (PS) were obtained by summing the scores in the TPS and the planning score in TPSP. Problem solving ability was again measured by the Test on Problem Solving in Programming (TPSP). The TPSP was marked twice for each individual to obtain the scores for TPSP (planning) and TPSP (programming).

The null hypotheses addressed in this study were stated below :

1. The relationship of the scores on LPT, TMT, and PS with TPSP (programming) tends to be zero.
2. There is no significant difference in the scores of programming abilitiy between the students of the following dichotomous variables :
   (a) high/low general ability,
   (b) gender,
   (c) home computer possession, and
   (d) high/low teacher-student ratio.

Interview

Interviews were also conducted in this research to supplement the quantitative method applied. These interviews may be help to understand the behavior of problem solving in programming in pre-college students. The interviews served to go deeper into the motivations of respondents and their reasons for responding as they did in the programming test. Each interviewee was asked to think aloud on a problem in the programming test. Questions were asked by the researcher following the think aloud process to validate the model of programming hypothesized. Three central themes surrounding the programming problem were discussed, namely question understanding; planning and coding; testing and reformulating. Detailed questions were stated in appendix H.

3.3 Subjects

The target population of the present study are students studying computer studies in secondary schools in Hong Kong. There were around 10,000 students studying the two year computer studies course. The course spent about 80 hours on BASIC programming instruction and practice occupying about half of the curriculum time resource. Students were required to design at least four programs. The length of each program ranged from about 30 to 70 lines. They were also required to complete an EDP project in BASIC. These programming assignments were usually required to be completed after school.

Pea and Kurland (1987) identified four distinct levels of programming proficiency to guide researchers to assess

students' programming expertise. They are program user, code generator, program generator and software developer. According to the descriptions mentioned above, the subjects of the present study might be classified as program generators.

Most students of the present study were in the age groups 15 to 18. 260 students were selected to participate the present study. They were chosen at the convenience of the researcher. The sample composed of twelve classes of students in ten secondary schools in Hong Kong. This sample size was about 2.60 percent of the target population. The details of the sample were summarized in Table 2.

Table 2 : Characteristics of subjects selected

Chatacteristic	Number in each group	
Sex	Male	Female
	164 (63.1%)	96 (36.9%)
Home computer	Possess	Do not possess
	151 (58.1)	91 (35.0%)
Teacher-student ratio	High	Low
	160 (61.5%)	100 (38.5%)

All schools selected in the present study were government subsidized. The hardware configurations were the same for all schools, each with eleven sets of microcomputers. Contents in the programming curriculum were guided by the Syllabus for Computer Studies in Hong Kong (CDC, 1986) and hence there was no doubt for the uniformity of the syllabus coverage for all schools participated the study.

3.4 Procedure

Once schools had been selected and agreed to participate the research, a testing schedule would be arranged by the computer studies teacher in the school. The total test time was around three and a half hours. Tests were conducted in normal computer studies' lessons in February and March. The five instruments were group-administered by the teachers in each of the target classes. The tests occupied six lessons. The tests were usually completed in 2 to 4 weeks' time.

Students' responses in the Language Proficiency Test, the Template Measurement Test and the Advanced Progressive Matrices were coded for computer input. Students' responses in the Test on Procedural Skills and the Test on Problem Solving in Programming were compared to idealized solutions and scored for percent correct. Students' programming solutions in the Test on Problem Solving in Programming were compared again to idealized planning schemes and scored for percent attained. The scores were coded for computer input. Students' background variables and planning behaviors recorded in the questionnaire on problem solving in programming were also coded for computer input.

3.5 Instruments

There were five instruments employed in this study. Four of them, including the LPT, TMT, TPS and TPSP, were designed by the researcher. The Advanced Progressive Matrices (APM) was used as a measure of general ability (Raven, 1958). A study of the norms of Hong Kong students of age groups 15 to 18 in the performance of the APM had been conducted by Li (1986). The alpha reliability of the instrument (28 items) was found to be 0.72. A pilot study on testing the reliability of the instruments had been conducted on November, 1987.

1. The Language Proficiency Test

The LPT was designed to measure students' proficiency in the language features in BASIC. Misconceptions of the language features in BASIC and PASCAL in pre-college students were considered in setting the test items (Putnam, Sleeman, Baxter & Kuspa, 1986; Bayman & Mayer, 1983; Pea, 1986; Mayer, 1979). It was composed of 24 five-option multiple choice questions. The test contained 3 subscales measuring the data, control and execution structures of BASIC (Chaudhary, 1985). Data structure dealt with input, output statements and variables. Control structure dealt with the control flow of the language. Execution structure dealt with the sequential operations of assignment statement and various functions defined in the language. The test covered all the primitives in the ANSI BASIC and all the language features required by the computer studies curriculum in Hong Kong (CDC, 1986). The primitives covered in the test were listed in Table 3. The exhaustive listing guaranteed the external validity of the instrument.

All the multiple choice questions were comprehension items asking students to predict output from a short program. The programs focused on simple language features as listed in Table 3. The alpha reliability of the Language Proficiency Test in the pilot study was found to be .67. Item analysis revealed that some items in the test had low item-total correlation value. The alpha reliability of the instrument was improved to .70 after 6 items with low item-total correlation were deleted. The final instrument consisted of 18 items with 18 as maximum score.

Table 3 : Language features in the Language Proficiency Test

Data structure	Control Structure	Execution Structure
PRINT	IF...THEN	LET (=)
PRINT TAB	GOTO	SQR, INT, ABS, RND
INPUT	FOR/NEXT	MID$, LEFT$, RIGHT$
READ/DATA/RESTORE	GOSUB/RETURN	+, LEN, STR$, VAL
Varaibles	STOP	ASC, CHR$
Array variables	END	REM

2. *The Template Measurement Test*

The TMT was designed to tap the repertoire of templates of the subjects. The test consisted of 24 five-option multiple choice questions. It had three sections. Each section had 8 questions assessing template repertoires ranging from low, intermediate, to high levels (Shneiderman & Mayer, 1979; Linn, Sloane & Clancy, 1987). Low level templates were composed of fewer statements performing simplier tasks. Intermediate level templates were composed of more statements

performing more difficult tasks. High level templates were longer and were more generally applicable to a variety of situations. Each question in the test had one theme in testing a template possessed by the subject. Most of the themes were reported in the literatures reviewed. Table 4 listed the contents of templates measured.

Table 4 : Contents in the Template Measurement Test

Low level	*Intermediate level*	*High level*
Concept of counter	Table printout	Binary search
Round off values	Find smallest value	"Flag" concept
Selection	Find largest value	Bubble sort
Validate numeric input	Sequential search	
Accumulator	Sum an array	
Validate string input	Reading a set of data	

Half of the multiple choice questions consisted of programs having blank statements requiring students to fill in the blanks by choosing suitable choice. This technique derived from Soloway and Enrlich's method of research in assessing template repertoire (1984). Another one third of the questions required subjects to select a correct program segment from five options which would perform the required function as specified in the stem. The rest questions asked the subjects to abstract the meaning of a given program segment. The alpha reliability of the instrument in the pilot study was found to be .62. Item analysis revealed that some of the items had low item-total correction. The alpha reliability coefficient of the instrument was improved to .72 after 6 items with item-

total correlation were deleted. The final instrument consisted of 18 items with a maximum score of 18.

3. The Test on Procedural Skills

The TPS was designed to measure the testing and reformulation skills of the subjects. The test consisted of two sections. The first part had five questions measuring the testing skills. The second part also had five questions measuring the reformulation skills. The five items on testing skills required students to suggest sets of test data to fully test a program. The testing skills involved in the test were boundary considerations, testing all possible cases, invalid inputs considerations, and the role of terminating indicators in test data input. These testing skills in programming were discussed in the literatures reviewed. The five items on reformulating skill required students to make changes to programs. Three of the questions had multiple choice questions to guide the subjects to discover the sources of errors. Another question required the subjects to debug from an erroreous output. The last question required the subjects to debug the program by dry run. The maximum number of points on the testing and reformulation subscales were both 18. The maximum point of the test was 36. The mean point and standard deviation in the pilot study were 21.72 and 5.73 respectively. The points ranged from 10 to 30. The alpha reliability of the instrument in the pilot study was found to be .65. After deletion of two items with low item-total correlation, one from testing subscale and another from reformulating subscale, the alpha reliability coefficient was improved to .68. The

maximum point of the test became 32 as defined in appendix G.

4. *The Test on Problem Solving in Programming*

The TPSP was designed to measure the programming ability and the planning skill of the subjects. The test had three problems. Each carried equal proportion of marks. The problem inventory of the test derived from three sources. One was adopted form a problem which the author intended to encourage students to use subroutines (McDonald, 1987). The problem could be solved without subroutines but its size would become very large. The second problem was modified from an example in a text on BASIC programming which stresses on structure programming (Bishop, 1983). The problem was difficult to write without advanced planning. The complexity of the problem could be reduced by decomposing the problem using the input, process, and output strategy. Array must be introduced in the problem solution. The third problem was an example in a text which serves as an exemplar in teaching stepwise refinement (Findlay & Watt, 1981). This problem was also used as a problem inventory in an empirical study investigating problem decomposition stategies in program design (Ratcliff & Siddiqi, 1985). The three problems were chosen under the criteria that common templates, planning, testing and reformulating skills are required. Templates like reading a set of data, sequential search, input validation, finding average and sorting were required in the problems. The maximum number of points on each question was 25. The mean points and standard deviations for the three questions were 17.97, 5.05; 12.44, 6.13; and 11.87, 5.88

respectively. The alpha reliability of the instrument in the pilot study was found to be .78. A questionnaire was also conducted to survey the planning behaviors and difficulty of the questions in the Test. In a five point scale with 5 as the most difficult indicator, the mean responses in the pilot study on the three questions were 2.44, 3.33, and 3.21 respectivly. Among the 39 subjects, 13, 33 and 22 reported that they have preplanned their programming solutions in questions 1, 2 and 3 respectively. In considerating the long testing time of the instrument, question 3 was abandoned in the actual test. The maximun mark of each question in problem solving in programming was also changed to 33 as defined in the marking scheme in appendix G. The TPSP (Planning) was not measured in the pilot study.

3.6 Analysis Of Results

1. The reliability coefficients of the 5 instruments would be reported.

2. The Pearson correlation coefficient matrix for the LPT, TMT, PS, and TPSP scores would be reported.

3. Multiple regression analyses would be used to examine the contribution of the language proficiency variable, template possession variable, and procedural skill variables, first separately and later as an aggregate on programming. Finally, path analysis, which permits an interpretation of a postulated causal model would be used to tease out the direct and indirect effects of language proficiency, template, and procedural skills on programming.

4. Four one-way analyses of variance on LPT, TMT, PS and TPSP scores between the group having high general ability and the group having low; the two sex; the group possessing home computer and the group that do not; and, the group having high teacher-student ratio and the group having low.

5. Analyses of covariance on LPT, TMT, PS, and TPSP scores by the two sex, the group possessing home computer and the group that do not, and the group having high teacher-student ratio and the group having low with the advanced progressive matrices scores as covariate would also be reported.

CHAPTER IV RESULTS AND DISCUSSIONS

4.1 RELIABILITY OF THE INSTRUMENTS

The reliability coefficients of the 5 instruments used in this research were summarized in Tables 5 and 6. Table 5 reported the Cronbach alpha reliability coefficients. The fourth instrument was a Test on Problem Solving in Programming (TPSP) which contained two questions. These two questions were marked twice, one was for measuring students' planning skill, the other was for measuring students' programming performance. Their corresponding reliability coefficients were reported in Table 5 as TPSP (Planning skill) and TPSP (Programming) respectively.

Table 5 : Cronbach alpha reliability coefficients

Instruments	No. of questions	Cronbach alpha
1. LPT	18	.70
2. TMT	18	.71
3. TPS	8	.73
4. TPSP (Planning skill)	2	.50
TPSP (Pprogramming)	2	.64
5. APM	28	.72

The Cronbach alpha reliability coefficients for all instruments were substantially high except the TPSP. The reliability coefficients for TPSP (Planning skill) and TPSP (Programming) were moderately low but these were not totally unexpected as the intention of the design of the test. The first problem was designed to measure programming ability which was more inclined to the syntactical knowledge and at the same time less planning skill was required. The second

intended to measure programming ability requiring syntactical knowledge and more planning skill. In order to explain the different nature of problem solving in programming, whether requiring more on planning skill or depending more on syntactical knowledge, some of the results in the following sections would be explained using results on individual question in this test.

Table 6 summarized the Guttman and the equal length Spearman-Brown split half reliability coefficients of the tests. Since the TPSP had two questions only, its split half reliability coefficient was not reported. After matching the items pairwise, based on the item content and the item difficulty level, the tests were splitted into two halves. The item numbers in the two halves for the four instruments were reported in Table 6.

Table 6 : Split half reliability coefficients

	Item Numbers			Spearman-
Instruments	First Half	Second Half	Guttman	Brown
1. LPT	1,2,3,7,9, 11,13,14,15	5,6,4,8,10 12,18,17,16	.74	.75
2. TMT	1,2,3,6,7, 9,11,15,16	5,8,4,10,12 13,14,17,18	.70	.70
3. TPS	1,2,5,7	4,3,6,8	.81	.81
4. APM	1,3,7,13,11, 15,5,17,19, 23,26,27,25 21	4,2,6,8,9, 12,14,10,16, 18,22,20,24 28	.70	.70

The reliability coefficient of the twenty eight items of the Advanced Progressive Matrices (APM), Set II (Raven, 1958)

for this study was found to be .72 (Cronbach alpha). The value of split-half (Equal Length Spearman-Brown) was .70. Li (1986) used the 28 items APM test on Hong Kong students of age groups 15 to 18, the same age groups of the present study, and found the values of alpha and split-half to be .80 and .72 respectively which was close to what had been found in this research.

The substantially high reliability coefficients revealed that high internal consistency existed in these instruments.

4.2 COGNITIVE COMPONENTS RELATED TO PROGRAMMING

The hypothesized cognitive components related to programming in this study were language proficiency, template possession and procedural skills. The Pearson correlation coefficient matrix between all pairs of variables were shown in table 7. Table 7 revealed that all hypothesized cognitive components were significantly related to programming performance. The stated null hypothesis that there is no relationship between these cognitive components and programming performance can therefore be rejected. The correlations between programming performance and language proficiency, template possession and procedural skills were 0.69, 0.67 and 0.85 respectively indicating that they had 47.61%, 44.89% and 72.25% variance in common with programming performance. Among them, procedural skills had the largest proportion of variance in common with programming performance reflecting that procedural skills were most related to programming.

It could also be deduced from Table 7 that procedural skills correlated substantially with language proficiency and template possession. Language proficiency correlated moderately with template possession. These substantial correlations reflected that there were intimate relationships among the cognitive components. These relationships would be further investigated in this research.

Table 7 : Correlations among cognitive components and programming

Variables	1	2	3	4
1. Language proficiency	1.00			
2. Template possession	.64***	1.00		
3. Procedural skills	.63***	.64***	1.00	
4. Programming performance	.69***	.67***	.85***	1.00

*** p < .0001

Language Proficiency And Programming

The language features studied in this research consisted of three subscales, namely, data structure, control structure and execution structure. The Pearson correlation coefficients of each of these subscales with programming performance were summarized in Table 8.

Table 8 : Correlations among substructures in a programming language with programming performance

Variables	1	2	3	4
1. Data structure	1.00			
2. Control structure	.43***	1.00		
3. Execution structure	.41***	.29***	1.00	
4. Programming performance	.60***	.48***	.50***	1.00

*** p < .0001

Among the three subscales, data structure correlated most strongly with programming performance. The other two substructures, execution and control, correlated moderately with programming. Scheffe' multiple comparisons showed that students' performances among the three substructures were statistically different ($F(2,777) = 4.20$, $p < 0.05$). Table 9 showed the mean scores of the three substructures. It revealed that Form 5 students in Hong Kong scored the highest in the language features related to the data structure such as the input, output statements and the use of variables and array variables. They performed the weakest in the language features related to the control structures in BASIC such as IF/THEN...GOTO, GOTO, FOR/NEXT and GOSUB statements. The performance related to execution structure lied in between the data structure and the control structure.

*Table 9 : Comparison of mean scores in data, execution and control structure in the language proficiency test**

	Data structure	Execution structure	Control structure
Mean (S.D.)	4.19 (1.47)	3.94 (1.52)	3.84 (1.18)

*The maximum point in each subscale is 6.
Groups in the same line are statistically homogeneous groups.
Number of students in each group is 260.

These results revealed that grasping fundamental data structure is the most prominent factor in elementary programming. Students learning the first programming language performed the best in this aspect. The better they learned about the input, output statements and the syntactical meaning of variable and array variable, the better they performed in elementary programming. Control structure was difficult to students learning the first programming language. However, it was not so highly related to programming performance as the other two substructures. This phenomenon may be explained in the following way. Different programmers can utilize mix of control components to write a successful program. For example, if a student fails to apply a compound conditional to make control, it can be replaced by two or more IF/THEN statements each with a single conditional. Hence, it is reasonable to postulate that when a threshold amount of control structure knowledge is grasped, students can solve elementary programming problem with no further hinderance. However, the fluency in the application of the control flow

knowledge will be prominent to complicated programming. It's role may become dominant in senior programming.

Template And Programming

Template in this research was classified into three levels, low, intermediate and high. The Pearson correlation coefficients of each of these three levels with programming performance were summarized in Table 10.

Table 10 : Correlations among different levels of template with programming

Variables	1	2	3	4
1. Low level template	1.00			
2. Intermediate level template	.43***	1.00		
3. High level template	.40***	.37***	1.00	
4. Programming ability	.58***	.49***	.47***	1.00

*** $p < .0001$

Among the three levels, low level template possession correlated most strongly with programming performance. The other two levels, intermediate and high, correlated moderately with programming. The correlation coefficients occurred in a decending trend from low to high levels. Scheffe' multiple comparisons showed that students' performances among the three levels were statistically different ($F(2,756) = 52.80$, $p <$

0.0001). Table 11 showed the mean scores of the three levels. It was seen that Form 5 students in Hong Kong perform better in intermediate level and low level templates than high level. In other words, after a two year programming course, students in Hong Kong generally have grasped some elementary programming concepts such as counter and accumulator. They also have abstracted some chunks of codes in performing some commonly encountered tasks such as summing an array, finding the largest and smallest value among a set of values, and searching for a target value sequentially. However, they showed weaker abstraction on larger chunks of codes like sorting and binary search. They also showed weaker performances on the concept of "flag" in programming.

*Table 11 : Comparison of mean scores in low, intermediate and high level template**

	Intermediate level template	Low level template	High level template
Mean (S.D.)	4.06 (1.39)	3.98 (1.36)	2.89 (1.54)

*The maximum point in each level is 6.
Groups in the same line are statistically homogeneous groups.
Number of students in each group is 253.

These results revealed that students perform equally well in the abstraction of low and intermediate level templates. The low, intermediate and high level template abstraction had 33.64%, 24.01% and 22.09% variances in common with programming performance. This result indicated that low level template

abstraction plays a significant role in solving elementary programming problems and students perform quite well in this aspect in a first programming course. It was not suprising that low and intermediate level template abstraction have more variances in common with programming performance because they are fundamental knowledge in solving elementary programming problems.

Language And Template

There were two types of knowledge investigated in this study, namely, language proficiency and template possession, in relation to programming performance. Table 12 summarized that students' performance in these two aspects were statistically different ($F(1,511) = 13.17$, $p < 0.001$).

*Table 12 : Comparison of mean scores in language proficiency and template measurement tests**

	Language proficiency	Template possession
Mean (S.D.)	11.97 (3.20)	10.93 (3.32)
No. of subjects	260	253

* The maximum point is 18 in each test.

Students performed better in the language proficiency test than the test on template possession. Such a result get in line with the common postulation that template abstraction depends on the proficiency of the language features. Students good at language features may not necessarily good at template abstraction. However, students good at template abstraction should be good at language features. Hence, students in

general should perform better in language features than template abstraction. The result in this research did not reject this postulation.

Procedural Skills And Programming

Table 13 revealed that all hypothesized procedural skills were significantly related to programming performance.

Table 13 : Correlations among procedural skills and programming

Variables	1	2	3	4
1. Testing skill	1.00			
2. Reformulation skill	.46***	1.00		
3. Planning skill	.43***	.58***	1.00	
4. Programming ability	.52***	.70***	.79***	1.00

*** $p < .0001$

The correlations between programming performance and planning skill, reformulating skill and testing skill were .79, .70 and .52 respectively indicating that they had 62.41%, 49.00%, and 27.04% variances in common with programming performance. Among the three skills, planning skill had the largest variance in common with programming performance, reformulating skill came second and testing skill had the least.

In the test on problem solving in programming, a self-

reported survey on preplanning activities in solving the two programming problems were conducted. Amount the 238 respondents in question 1, 89 (37.39%) reported that they had preplanned the programming solution. Amount the 205 respondents in question 2, 52 (25.37%) reported that they had preplanned. An analysis of variance between the students who reported preplanning and those who do not on programming scores in question 1 ($F(1,236) = 4.507$, $p < .05$) and 2 ($F(1,203) = 5.376$, $p < .05$) indicated that students preplanning the programming solutions did significantly better than those who did not preplan. These results revealed that preplanning activity was not common, only about one quarter to one third of the students preplanned their programming solutions, but preplanning favourited programming performance.

4.3 A PATH MODEL ON PROGRAMMING

To examine the contribution of language proficiency, template possession and the three procedural skills to problem solving in programming, stepwise multiple regression analyses were performed. To tease out the direct and indirect effects of these interrelated components on programming, path analysis was used as a method of decomposing and interpreting the linear relationships among the language proficiency score, template possession score and procedural skills score.

Multiple Regression Analyses

Before applying the multiple regression analyses, certain aptness test of the multiple regression model was tested

against suppression and multicollinearity (see Lo, 1982). It was found that the suppression concerned with the gain or loss of the portion of variance in the criterion variable, which was accounted for by the independent variables, due to high correlation between the independent variables, was not envidenced. In the test of multicollinearity, a small drop of the regression coefficient in the forward stepwise regression was observed. This might infer that small multicollinearity was observed.

The linear dependency of programming on the three procedural skills, namely, testing, reformulating and planning, and on language proficiency and template possession was shown in Table 14.

Table 14 : Linear dependency of programming on planning, reformulating, testing, language, & template scores

Rank	Variable	R	R^2	Increase in R^2	Simple r	Standard[a] error of estimate	F
1.	Planning	.787	.620	.620	.787	9.979	324.28***
2.	Language	.843	.711	.091	.693	8.715	244.01***
3.	Reformulation	.865	.748	.037	.701	8.159	195.23***
4.	Template	.871	.759	.011	.667	7.998	154.63***
5.	Testing	.877	.770	.011	.524	7.847	130.23***

[a]Criterion variable: Programming performance($\bar{X}$=34.33, SD=16.07)

*** $p < .0001$

Of the total explained variance of 77%, 66.8% was accounted for by the planning, reformulating and testing

tasks. This led to the use of these three tasks as an aggregate together with language proficiency and template possession in a further stepwise multiple regression analysis. The results were shown in Table 15. Procedural skills, language and template emerged as significant variables in programming and accounted for 76.10% of the variance. To make explicit the theoretical postulate that template and procedural skills affects problem solving in programming more than language proficiency, path analysis was used as a strategy of further analysis.

Table 15 : Linear dependency of programming on procedural skills, language and template scores

Rank	Variable	R	R^2	Increase in R^2	Simple r	Standard[a] error of estimate	F
1.	Procedural skills	.846	.715	.715	.846	8.637	499.52***
2.	Language	.867	.752	.037	.693	8.075	300.55***
3.	Template	.872	.761	.009	.667	7.959	208.52***

[a]Criterion variable: Programming performance($\overline{X}$=34.33,SD=16.07)

*** p < .0001

Path Model Analyses

The path models were displayed in Figures 9 and 10. The displayed models are simple, recursive, just-identified path models, drawn from presumed causes (exogenous variables) to presumed effects (endogenous variables). The LISREL procedure in SPSSX package (Nie, 1983) was used for the path model

analyses. The computational results were summarized in the path diagrams in Figures 9 and 10.

Figure 9 displayed a simplified model explaining students' programming performance as a product of language proficiency and design skills. Design skills score was the sum of the raw scores of template and procedural skills. This model was based on Linn's theoretical chain of cognitive requirement in programming (Linn, 1985). This model accounted 50.50% of the variances.

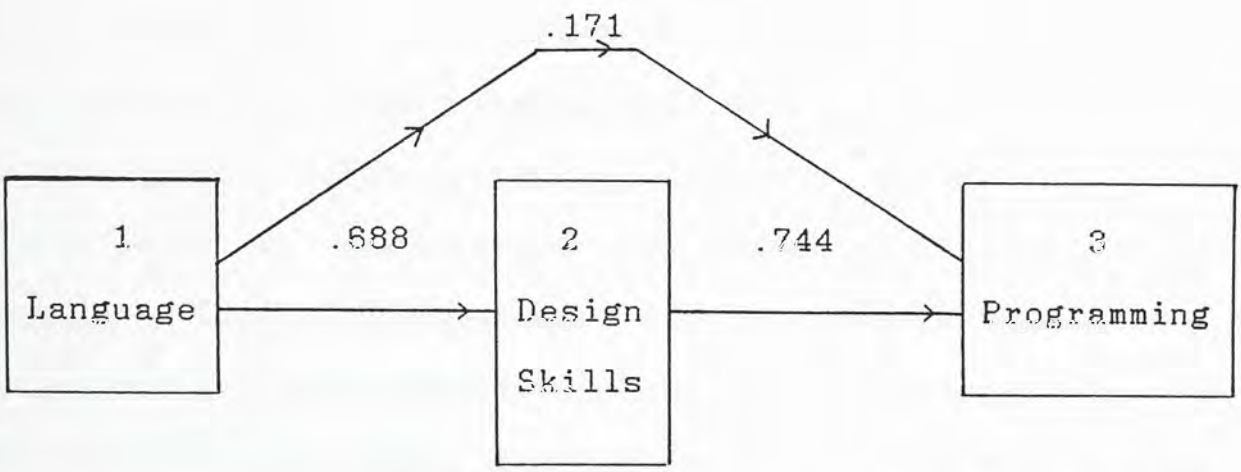
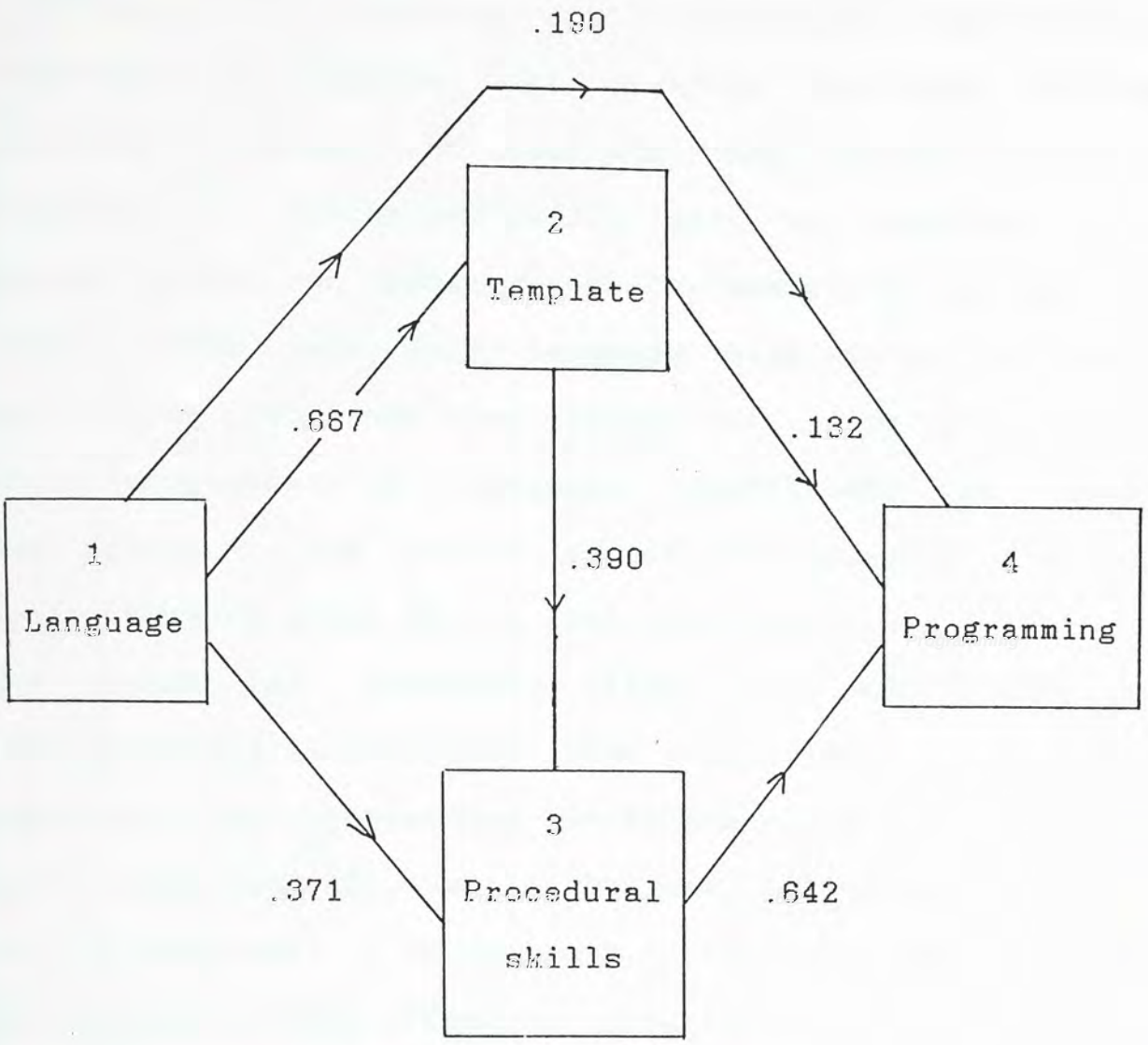


Figure 9 : Schematic path diagram of language proficiency, design skills and programming.

The model in Figure 9 served to answer three hypotheses in this study. The first hypothesis was that language proficiency would exert a direct effect on programming. As shown in Figure 9, the path coefficient for this effect was .171. These results suggested that language proficiency did exert a direct and unique effect on programming. That is, higher language proficiency is associated with better

programming performance. The second hypothesis was that language proficiency would exert an indirect effect on programming. As shown in Figure 9, this indirect effect was hypothesized to be mediated by the design skills required in programming. The path coefficient from language proficiency to design skills was .688 indicating that there existed an intimate relationship between language proficiency and design skills. Figure 9 also showed that the path coefficient for the effect of design skills on programming was .744. Therefore, the indirect effect of language proficiency mediated through design skills on programming was .512 (.688x.744). The total effect of language proficiency (direct and indirect) on programming was .683 (.171+.512). It could be seen that a large proportion of the effect of language proficiency on programming is mediated through the design skills. The third hypothesis was that design skills affected programming independently. More specifically, as scores on design skills increase, programming scores increase. Figure 9 shows that this effect was .744.

Figure 10 presented a more detailed model explaining students' programing performance as a product of language proficiency, template possession and procedural skills. The design skill was separated into template possession and procedural skills as postulated by Linn. The path coefficients were shown in Figure 10. Table 16 summarized the path coefficients for the direct and indirect effects of the independent variables on programming performance. This path model accounted for 55.00% of the variance.



$R^2_{4.123} = .550$

Figure 10 : Schematic path diagram of language proficiency, template, procedural skills and programming.

Table 16 : Effects of language proficiency, template possession and procedural skills on programming

Independent Variable	Direct Effect	Indirect Effect	Total Effect
1. Language proficiency	.190	.493	.683
2. Template possession	.132	.250	.382
3. Procedural skills	.642	----	.642

Figure 10 displayed the solved path model to explain programming performance, with controls for both knowledge, including language and template, and procedural skills in programming. Procedural skills had, as expected, a very strong effect on programming (as symbolized by the path of .642). The path from language proficiency to template abstraction (.667) was also substantial, clearly illustrating the importance of language proficiency on template abstraction. The direct effect of language proficiency on programming as given by the path coefficient was .190, while the total of indirect effect was .493 (.667X.132 + .667X.390X.642 + .371X.642). The direct effect of template possession on programming performance as given by the path coefficient was .132, while the total of indirect effect was .250 (.390X.642). In contrast, the procedural skills had a much greater direct effect on programming as shown by the path coefficient of .642. Thus, within the framework of the postulated causal model, there was some evidence from the path analysis to support the assertion that the effect of language proficiency on programming is mediated by template abstraction and which in turn is mediated by the procedural skills and the latter has a greater direct effect on the deliberately acquired task of programming.

Path Model Analyses On Individual Programming Problem

Figures 11 and 12 showed the solved path models to explain the role of language, template and procedural skills on problem solving in programming when the requirement of the problem was differentiated. There were two questions in the

programming test. The first question required more syntactical knowledge and less planning skill, and whose path model was shown in Figure 11. The second question required more planning skill, and whose path model was shown in Figure 12. The path coefficients for the direct and indirect effect of the independent variables on programming for each question were summarized in Table 17 and 18.

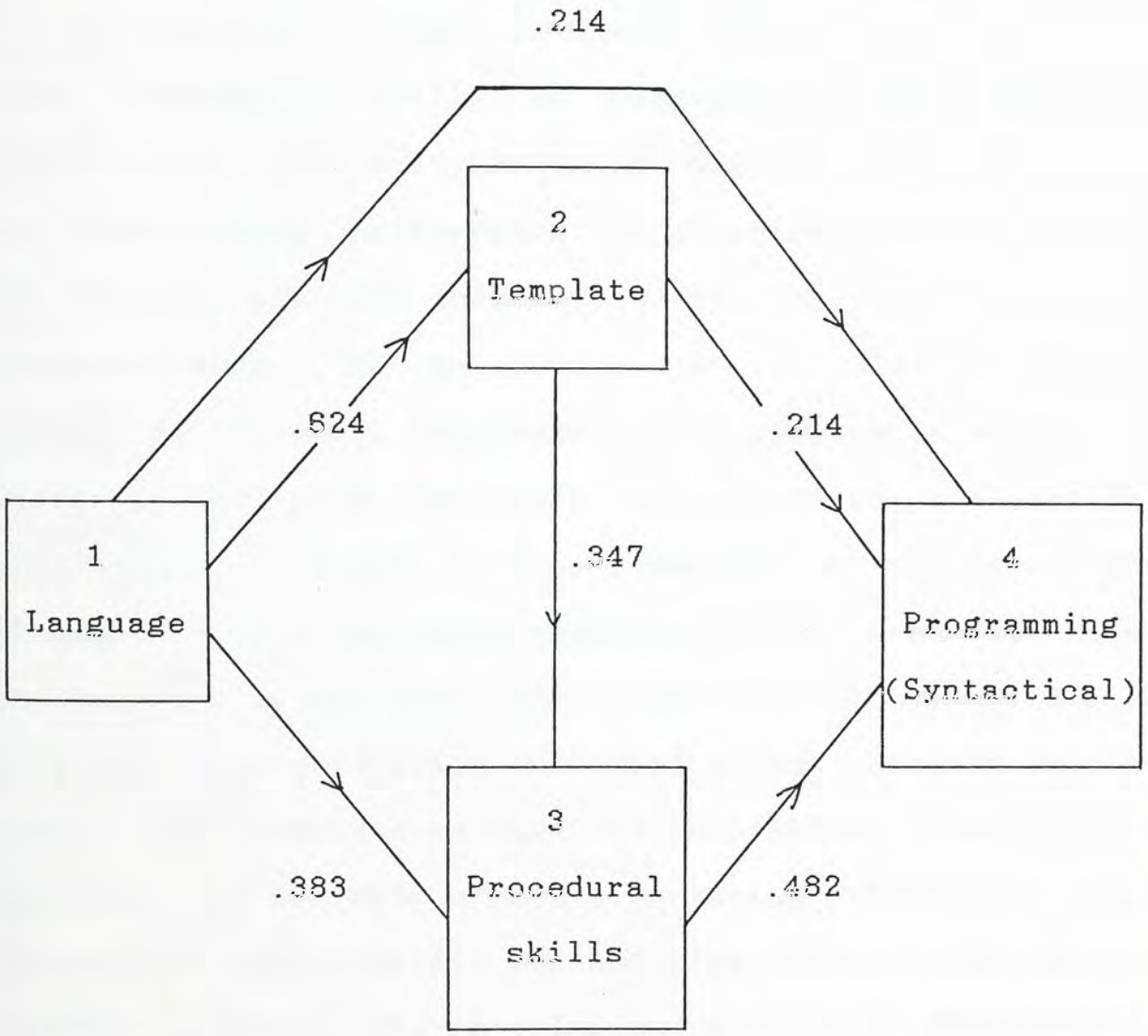


Figure 11 : Schematic path diagram of language, template, procedural skills and programming which depends more on syntactical knowledge.

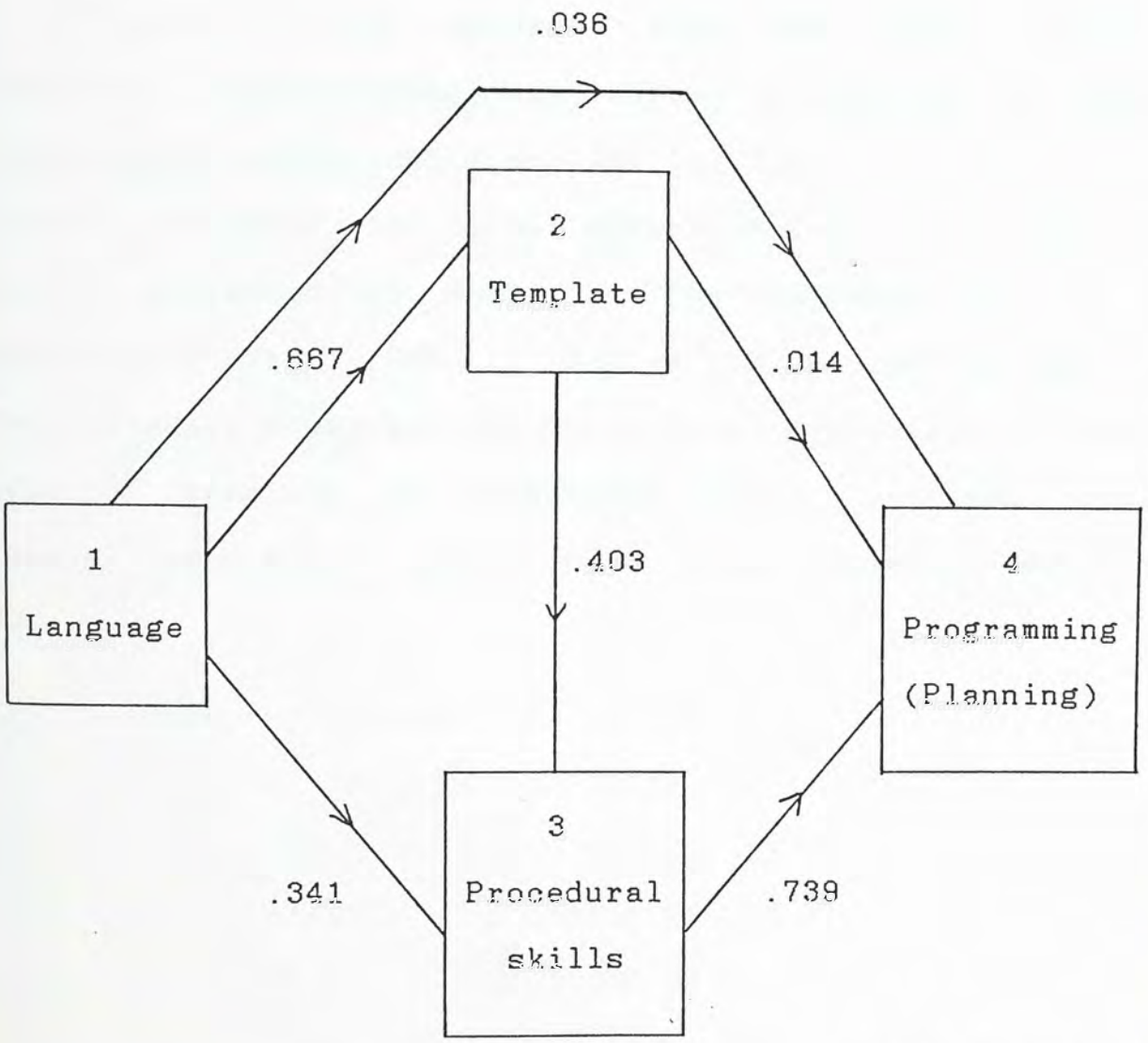
Table 17 : Effects of language, template & procedural skills on programming which depends more on syntactical knowledge

Independent Variable	Direct Effect	Indirect Effect	Total Effect
1. Language proficiency	.214	.429	.643
2. Template possession	.214	.167	.381
3. Procedural skills	.482	----	.482

As expected, Figure 11 showed that the path coefficient from procedural skills to programming was substantially reduced from .642 in Figure 10 to .482 in Figure 11 when only the programming performance in question one was considered. In contrast, the path coefficient from language to programming increased from .190 in Figure 10 to .214 in Figure 11 reflecting that the requirement of a problem is a significant factor in analysing the role of knowledge and skills in programming. Figure 11 also revealed that the direct effect of template on programming increased which was reflected by the increase in the path coefficient from .132 in Figure 10 to .214 in Figure 11 while the total effect remained nearly the same. This result supported the theoretical postulation that template may not only affect programming performance mediated through procedural skills but may also affect it directly when suitable templates are recalled and applied in programming.

As expected, Figure 12 showed that the path coefficient from language to programming was substantially reduced from .190 in Figure 10 to .036 in figure 12 when only the programming performance in question two was considered. In

contrast, the path coefficient from procedural skills to programming increased from .642 to .739 reflecting that procedural skills and especially planning skill is predominant in the question. These results once again confirmed that the requirement of a programming problem will affect the role of knowledge and skills involved in it.



$R^2_{4.123} = .506$

Figure 12 : Schematic path diagram of language, template, procedural skills and programming which depends more on planning skill.

Table 18 : Effects of language, template & procedural skills on programming which depends more on planning skill

Independent Variable	Direct Effect	Indirect Effect	Total Effect
1. Language proficiency	.036	.460	.496
2. Template possession	.014	.298	.312
3. Procedural skills	.739	----	.739

Figure 12 also revealed that the direct effect of template on programming was reduced as shown by the decrease in the path coefficient from .132 in Figure 10 to .014 in Figure 12 while the direct effect of template on procedural skills increased as shown by the increase in the path coefficient from .390 in Figure 10 to .403 in Figure 12. These results supported the theoretical postulation that the role of template on progamming will increase mediated through procedural skills when more planning skill is required.

4.4 INDIVIDUAL DIFFERENCE ON PROGRAMMING

One of the objective of this research is to investigate the relations between individual difference variables and students' achievement in introductory programming courses. The analysis presented in this section focused on the associations between student characteristics that existed and student performance on various programming components that postulated in this study, namely, language proficiency, template possession, testing skill, reformulating skill, planning skill and problem solving in programming. Four sets of students characteristic that were expected to affect student success in programming were hypothesized. The sets of student characteristics were general ability, gender, home computer possession and teacher-student ratio in the programming classes.

Two set of analyses were conducted to examine the individual differences questions. First, group comparisons (F-tests) of individual difference variables were conducted on programming performances. Because of the research findings in various reports (Linn, 1985; Dalbey, & Linn, 1986; Fisher & Mandinach, 1985) indicated that general ability clearly influence programming performances, further analyses of covariance using the APM score as covariate would be conducted when the individual difference variables, namely, gender, home computer possession, and teacher-student ratio, were found to be significantly related to the programming tasks in the first set of analyses.

General Ability

The APM score was used as a measure of general ability in this study. The mean and the standard deviation of the APM score were found to be 18.88 and 3.83 respectively. Since one of the school could not conduct the APM test in time, the total number of subjects in the analysis was 215 only. The subjects were classified as high and low in general ability when their APM scores were above or below the mean APM score respectively. Analyses of variances on general ability for the programming components indicated significant differences.

Table 19: Analysis of programming components by general ability

Component	Mean		*SD*		F
	High	Low	High	Low	
Language Proficiency	13.12	10.74	3.02	3.18	31.62***
	(113)	(102)			
Template Possession	11.96	9.45	3.05	3.11	35.73***
	(113)	(102)			
Testing Skill	12.43	11.55	2.20	2.20	8.24**
	(111)	(96)			
Reformulating Skill	7.78	5.00	4.19	4.01	23.67***
	(111)	(96)			
Planning Skill	9.00	4.95	5.77	4.03	27.05***
	(96)	(76)			
Programming	39.75	26.17	14.97	14.71	35.43***
	(96)	(76)			

** $p < .01$. *** $p < .001$

Number in the brackets indicates the number of subjects in that group.

An examination of Table 19 revealed that there were sizable differences between general ability on all of the programming components. Students classified as high in general ability performed consistently better than students classified as low in general ability in all programming tasks. The null hypothesis that there is no significant differences in the programming performance between high/low general ability could be rejected. Since the differences between the two groups were large, correlations between the APM score and all the programming components were further explored.

Table 20 : Correlations between APM score & scores in programming tasks

	APM Score
Language proficiency	.41***
Template possession	.45***
Testing skill	.20**
Reformulating skill	.34***
Planning skill	.35***
Programming	.42***

** p < .01 *** p < .001

Table 20 revealed that all programming components were significantly related to the APM score. The Pearson correlation coefficients were found in the range of .20 to .45 indicating that the APM test score had 4.00% to 20.25% of the variances in common with the programming performances. These results were similar to the findings of Linn & Dalbey (1985),

and Fisher & Mandinach (1985). Their correlations of ability and success in programming in general ranged between .15 to .54. Among the programming components, general ability measured by the APM score was found to be most related to template abstraction, programming problem solving and language proficiency. Procedural skills were found to be consistently less related to general ability.

These results suggested that procedural skills were more learnable and depended less on general ability. On the contrary, template abstraction, programming and language learning depended more on general ability but their overall variances in common with general ability were still low indicating that general ability is just one of the factors affecting programming learning. General ability appeared to be helpful but not sufficient for the betterment of programming learning. Form of instruction had been found to interact with general ability in programming performance. Linn & Dalbey (1985) and Mandinach & Fisher (1985) found that there was a direct relationship between ability and programming performance in schools where instruction emphasized language features. On the contrary, in schools where instruction emphasized explicit program design, high and medium ability students performed equally well.

Gender

Analyses of variance on each of the programming components between students of opposite sex indicated some significant differences. An examination of Table 21 revealed that male students performed significantly better than female

students in language proficiency and reformulating skill. Template possession, testing skill, planning skill and problem solving in programming had no statistical differences between the two sex.

Table 21 : Analysis of variance on programming components between students of opposite sex

Component	Mean		*SD*		F
	Male	Female	Male	Female	
Language Proficiency	12.52 (164)	11.04 (96)	3.29	2.81	13.56***
Template Possession	11.04 (164)	10.73 (89)	3.36	3.25	0.49
Testing Skill	12.16 (155)	12.14 (91)	2.20	2.27	0.00
Reformulating Skill	7.24 (155)	5.46 (91)	4.36	4.18	9.81**
Planning Skill	7.71 (121)	7.78 (92)	5.82	4.93	0.01
Programming	35.45 (121)	32.86 (92)	17.00	14.71	1.37

** p < .01. *** p < .001

Number in the brackets indicates the number of subjects in that group.

In order to make sure that the differences were not affected by ability differences, analyses of covariances were performed. Results revealed that the gender differences on language proficiency ($F(1,214) = 17.29$, $p < .001$) and

reformulating skill ($F(1,206) = 15.79$, $p < .001$) still existed and that the differences were not due to intelligence.

Many hypothesized that male students perform better than female students in programming performance. Results from the present research did not fully support this hypothesis. There were no gender differences in template abstraction, testing skill, planning skill and programming. Male students performed better only in language proficiency and reformulating skill. Results from studies of middle-school programming also indicated that there were no gender differences in achievements from instruction in BASIC (Linn & Dalbey, 1985; Mandinach & Fisher, 1985). Webb also reported that there were no gender differences in outcomes measured on learning LOGO at the middle-school level (Webb, 1984). On the other hand, gender differences favouring male students had been found for computer access and interest in computer (Mandinach & Linn, 1986). Language proficiency and reformulating skill are directly related to computer access and interest because language features and debugging skills can be learnt from practising and time devoted. This may explain why male students outperformed female students in these two aspects.

Home Computer Possession

Analyses of variances on programming components between students with and without home computer revealed that there were no statistical significant differences. Table 22 indicated that students possessing home computer performed slightly better than those students who do not possess home

computer in all programming components except the testing skill. The differecnes were not statistically significant. These results did not support the common hypothesis that students possessing home computer would perform better than those do not.

Table 22 : Analysis of variance on programming components between students with and without home computer

Component	Mean		*SD*		F
	Yes	No	Yes	No	
Language Proficiency	12.29 (151)	11.75 (91)	3.29	3.02	1.65
Template Possession	10.95 (149)	10.80 (87)	3.23	3.47	0.11
Testing Skill	12.04 (145)	12.15 (85)	2.25	2.21	0.13
Reformulating Skill	6.85 (145)	6.05 (85)	4.31	4.55	1.78
Planning Skill	7.79 (127)	7.39 (69)	5.55	5.50	0.23
Programming	34.61 (127)	32.70 (69)	16.38	16.37	0.61

Number in the brackets indicates the number of subjects in that group.

The mean class size for the 10 schools studied in these research was 24.83 and the class size ranged from 13 to 38 students. There were 11 sets of microcomputers in each school. The student-machine ratio was approximately two to one.

Students were usually allowed to access the computers for one to two hours per school days. It could be concluded that the school computer access time in the settings of Hong Kong are extensive. The mean programming score in this research was 34.33 (maximum score was 66). It could be said that students studying programming for two years time have made them to go far enough the chain of cognitive accomplishments and that their concern is with design of programs. These results supported the hypothesis postulated in this research that when access is extensive and when students are concerning with programming design, then the influence of out-of-school access is small. Linn & Dalbey (1985) and Fisher & Mandinach (1985) found similar results in their studies of programming performance in BASIC at the middle-school level.

Teacher-student Ratio

There were two types of programming classes investigated in this study, one had high teacher-student ratio, the other had low teacher-student ratio. Most schools helding programming courses in Hong Kong had eleven sets of microcomputers. Therefore those schools had less than or equal to 22 students in one class were classified as high teacher-student ratio classes. Those schools had more than 22 students in a class were categorized as low teacher-student ratio classes. Analyses of variances on the programming components between classes with high and low teacher-student ratio revealed some significant differences.

Table 23: Analysis of variance on programming components between classes with different teacher-student ratio

Component	Mean		*SD*		F
	High	Low	High	Low	
Language Proficiency	11.46	12.80	3.05	3.27	11.31***
	(160)	(100)			
Template Possession	10.84	11.06	3.46	3.11	0.26
	(153)	(100)			
Testing Skill	12.25	12.01	2.24	2.20	0.66
	(150)	(96)			
Reformulating Skill	5.95	7.57	4.29	4.34	8.34**
	(150)	(96)			
Planning Skill	7.20	9.06	5.45	5.22	5.27*
	(151)	(62)			
Programming	32.25	39.42	16.24	14.55	9.10**
	(151)	(62)			

* $p < .05$ ** $p < .01$ *** $p < .001$

Number in the brackets indicates the number of subjects in that group.

Table 23 revealed that students in low teacher-student ratio classes performed consistently better than those in high teacher-student ratio classes in all programming components except the testing skill. Among them, only language proficiency, reformulating skill, planning skill and programming were statistically significant. It is generally assumed that students who attend classes with high teacher-student ratio would perform better than those attending classes with low teacher-student ratio. The results in this

study were not consistent with this common postulation. Further analyses of covariances using APM score as covariates were performed and were summarized in Tables 24, 25, 26, and 27. These results revealed that the teacher-student ratio differences on language proficiency, reformulating skill, planning skill and programming existed and that the differences were not due to general ability.

Table 24 : Analysis of covariance on language scores by teacher-student ratio with the APM score as covariate

Source	Sum of Squares	df	Mean Squares	F
Teacher-student ratio	58.08	1	58.08	6.47*
Error	1902.59	212	8.97	

* p < .05

Table 25: Analysis of covariance on reformulating skill scores by teacher-student ratio with the APM score as covariate

Source	Sum of Squares	df	Mean Squares	F
Teacher-student ratio	103.49	1	103.49	6.39*
Error	3303.61	204	16.19	

* p < .05

Table 26 : Analysis of covariance on planning skill scores by teacher-student ratio with the APM score as covariate

Source	Sum of Squares	df	Mean Squares	F
Teacher-student ratio	133.85	1	133.85	5.25*
Error	4304.90	169	25.47	

* p < .05

Table 27 : Analysis of covariance on programming scores by teacher-student ratio with the APM score as covariant

Source	Sum of Squares	df	Mean Squares	F
Teacher-student ratio	1015.40	1	1015.40	4.71*
Error	36449.13	169	215.68	

* p < .05

It is reasonable to argue that higher teacher-student ratio may not necessarily bring along better programming performance unless appropriate measures, like method of instruction, are incorporated. Though teacher-student ratio has been raised in programming course in Hong Kong in recent years, no special measures, like teaching training and improvement in method of instruction, have been adopted to improve the teaching effectiveness. Such a result was not totally unexpected. There may be other factors which were not investigated in this research, like teachers' programming teaching experiences and learning atmosphere in low teacher-student ratio classes, that contribute to the differences. No concrete reason accounting for the differences can be made in this study.

Interactions

Since general ability had a moderately high correlation with programming performance, only those individual variables had significant effects on programming performance after teasing out the effect of general ability would be further investigated. Under such a criterion, the interaction

effects of sex and teacher-student ratio on language proficiency and reformulating skill were analysed. When 2-way interactions of sex and teacher-student ratio on language proficiency and reformulating skill were analysed, no significant differences were found. The statistics were summarized in Table 28, 29, 30 and 31.

Table 28:Analysis of covariance on language proficiency scores by sex & teacher-student ratio with the APM score as covariate

Source	Sum of Squares	df	Mean Squares	F
Sex	103.68	1	103.68	12.11***
Teacher-student ratio	13.88	1	13.88	1.62
Sex X Teacher-student ratio	0.67	1	0.67	0.08
Error	1798.24	210	10.98	

*** p < .001

Table 29:Analysis of covariance on reformulating skill scores by sex & teacher-student ratio with the APM score as covariate

Source	Sum of Squares	df	Mean Squares	F
Sex	168.90	1	168.90	10.94***
Teacher-student ratio	27.60	1	27.60	1.79
Sex X Teacher-student ratio	17.04	1	17.04	1.10
Error	3117.67	202	15.43	

*** p < .001

Table 30 : Comparison of mean scores on programming components by sex with APM score as covariate

Components	Male	Female
Language proficiency	12.53	10.37
	(161)	(54)
Reformulating skill	7.21	4.40
	(154)	(53)

Number in the brackets indicates the number of subjects in that group.

Table 31 : Comparison of mean scores on programming components by teacher-student ratio with APM score as covariate

Components	High	Low
Language proficiency	11.28	12.80
	(115)	(100)
Reformulating skill	5.56	7.57
	(111)	(96)

Number in the brackets indicates the number of subjects in that group.

4.5 FINDINGS FROM INTERVIEW

The results presented in this section based on three programming protocols. The protocols were derived from one and a half hour interviews of three voluntary students recommended by their teachers. The interviewees were requested to think aloud in solving a programming problem. The process was tape recorded for analysis. Structure questions as shown in Appendix H were then asked to probe further details about the problem solving process. The protocols and the subjects' programming solutions were recorded in Appendix I.

Planning Behavior

In the interviews, the students showed a common planning behavior in solving the programming problem. Each of them formed a rough plan in mind after they understood the requirements of the problem. The following was a typical sample rough plan.

> "I shall think of myself as a cashier. There is someone coming to buy things for certian quantities. This information will be entered into the computer. Prices must then be searched from the data table. A searching is necessary... Calculate total. Ask for cash. Calculate change. Ask for cash again if not sufficient money is given. Output can be done easily by following the sample output. The price, stock code, and quantity should be printed out on the bill."

However, these rough plans were not refined to concrete programming steps before coding. For example, how the control flow could be passed after entering all stock codes; when the program would be ended; should arrays be introduced to store stock codes or prices for later usage. They started to code from the rough plans without further refinement. Among the three subjects, two of them spontaneously said that no flowcharting was required in the question and they won't consider to use this planning technique. When they were asked whether they used flowchart as planning tool in their programming assignments, they said they wouldn't use it unless it was required by their teachers. It should be noted that flowcharting had been taught in their two year programming course as the major planning tool. The remaining subject said that she uses flowchart sometimes, but usually only the control flow of a certain part of the program will be constructed. This subject did not use flow diagram to plan too in the interview.

Because no concrete planning was made before coding, all of them had to revise their plans in the programming process. Usually, they spent quite a lot of time in revising their plans. For example, in solving the programming problem in the interview, all of them did not use an array to store the price of stocks brought by the customer in the program until they need it in calculating the total cost or printing the prices on the bill. Another typical example was that all of them added the statment to compare the stock code and "BYE" to check the end of the program when they felt the program should be ended. They were typical problem solvers using current

decisions and observations to make decisions about what to do next. This planning behavior was termed as planning-in-action or opportunistic planning by Pea and Kurland (1984). Webb and his colleagues (1986), Dalbey, Tourniaire & Linn (1986) had similar results in observing programming behavior in elementary programming courses.

Templates

In the interviews, the three students demonstrated clearly their repertoire of templates and the role of templates in their programming and planning behavior. All of them showed a certain level of template possession in their programming process. They applied low level templates like counting, summing total, and checking positive integers for inputted values quite smoothly demonstrating that they had abstracted these pieces of codes from experiences. Subject A showed the best in applying these low level templates. He added the statements like "M = 1" and "T = 0" immediately when M was counted and T was accumulated. Subject B had to think for a while before producing these codes. Subect C produced the codes for summing the total, "T = 0" and "T = T + COST(J)", correctly but the statements were put in inappropriate order. The researcher felt that the possession of these low level templates affect programming performance quite strongly.

In the interview, subject A clearly demonstrated how the codings on sequential search was produced from the abstract structure in his mind. Subject A produced the codings on sequential search as shown below.

```
210 INPUT "STOCK CODE ?";M$(M)
220 FOR I = 1 TO N
230 IF M$(M) = SC$(I) THEN GOTO 300
236 P(M) = PC(I)
240 NEXT I
250 PRINT "NO SUCH STOCK CODE" : GOTO 210
300 ...
```

When subject A wrote the codes, he wrote line 230 jumping to line 300 with no hesitation before line 236, 240 and 250 were produced. The researcher asked him why he coded the program segment this way immediately in the interview. He answered in the following manner.

> "This line (230) means if M$(M) is found, then jump out the searching loop, therefore jump to line 300, else check continuously inside the loop. If not found at the end of loop, then it means M$(M) not in the data table, print "NO SUCH STOCK CODE" and jump to line 210 for reenter."

Subject A demonstrated that he had the sequential search algorithm in mind before coding. He was just applying his knowledge in BASIC to produce the codes in answering this part of the programming problem. It was observed that subject A just used three minutes to finish the coding. In contrast, subjects B and C did not possess this template, they spent much time on struggling for the codes on sequential search. However, both of them made serious mistakes on the searching

algorithm. Subject B could just determine whether the stock code was in the data table or not, but could not make use of the index to search price for the stock code. Subject C finally recognized that she could make use of the "flag" to store the index for finding the price after struggling for a very long period of time, but her algorithm would never be adopted by any porgrammer because she read the data table each time when a matching was compared. Similarly, the three subjects showed differences on the template in reading a set of data until a terminator was encountered. Though both subjects A and B could write the codes correctly, subject A demonstrated a better quality of codes by using counter N to perform the counting. Subject C had some idea to read, but she made a serious mistake in missing a GOTO statement for forming a loop to read.

These results validated that templates were not memorized in a line-by-line form, instead they were abstracted as a schema as postulated by Linn (1985) or as an internal semantic as proposed by Shneiderman & Mayer (1979). Templates may also assist the learner to code programming lines in an expert manner which will be useful for them to "steal" the codes in applying to different situations. It could be seen that codes generated by the programmers without guidance would not be generalizable to other application areas.

In the interviews, it was observed that templates played a significant role in planning. The rough plans produced by the subjects included low and intermediate level templates like calculate total, and searching, etc. If subjects A and B did not possess these templates, they would not be so

confident in producing their rough plans. Subject C was a typical unsucessful problem solver among the interviewees. She knew searching and summing total were needed, but she could not integrated these parts into her programming solution smoothly because she didn't realize their inter-relationships. Subject C did not know a searching was to match a key and an array of elements, so she put the codes in reading an array and searching a key simultaneously. She knew the necessity of accumulating the total with the two statements "T = 0" and "T = T + COST(J)", but she put the order wrongly. "T = 0" must be put to an appropriate place in her program to sum. The following paragraph may illustrate why subject C could not plan properly. She said

> "This program is, in fact, difficult to us in comparison to the assignments and examination questions. Usually, the steps are given in examinations. We only need to follow the steps. It will be more easy. But in this programming problem, we find mistakes at the rear part of the program. We have to jump back to make corrections. In examination, it is certainly correct if we follow the details of each part."

It can be seen that if students are not given problems which require their own planning, they will certainly not be able to plan. They also will be unable to integrate their templates into programming plans.

Language Features

In the interviews, it was observed that all three subjects could express their programming idea in general by using the BASIC language features. Subject C performed the worst among the three in the programming. She had more problems on the language features. First, she suspected whether a variable could be added to an array variable or not in setting "T = T + COST(J)". Second, she misunderstood that Q(J) could store string value and tried to find the value of Q(J) by VAL(Q(J)) in her programming solution. Third, she suspected that the variable FLAG would store the value of N at the end of the following loop :

```
70 FOR I = 1 to N
90 IF SC$(I) = CSC$(I) THEN FLAG = I
100 NEXT I
```

This suspection reflected that the subject was confusing about the value of the variable I inside the loop. She thought that the value of I in "FLAG = I" in line 90 would change as the value of the counter I change. These misconceptions and mistakes on the language features had hindered subject C's programming performance.

CHAPTER V SUMMARY, CONCLUSIONS AND RECOMMENDATIONS

5.1 SUMMARY OF FINDINGS

Instrument

The results cited above showed that the instruments, Language Proficiency Test, Template Measurement Test, Test on Procedural Skills and Test on Problem Solving in Programming designed in this research were reliable.

Cognitive Components Related To Programming

Language proficiency, template possession and procedural skills scores were found significantly related to programming performance. These cognitive components were found highly related to problem solving in programming both from the quantitative and qualitative analyses.

Among the three subscales in language proficiency, students performed better in data and execution structure than control structure, and students' performance in data structure was found most related to programming performance. Results from the programming protocol analysis also revealed that students weak in handling language features had more problems in programming.

Among the three level of templates, students performed better in low and intermediate level than high level, and students' performance in low level template was found most related to programming performance. The interviews also reflected that students did make use of their repertoire of templates in programming. The better they abstrated the

templates, the better were the programming performance. These research also found that students performed better in the language proficiency test than the template possession test.

The three procedural skills, planning, testing and reformulating, were found highly related to programming performance. Planning score was most related to programming score, reformulating score came the second. A survey on preplanning behavior indicated that students who preplan their programs performed significantly better than those who did not. However, there were only about one quarter to one third of the students in the research claimed that they had preplanned their programs. The programming protocol analysis revealed that the planning activities were usually in the form of a rough plan in mind only. The plans were not refined to explicit programming steps. Flowcharting was an unfavourable planning tool and was seldom used. The planning behavior in the interviews were observed as opportunistic and much time was spent on revising the plans. The protocol analysis also revealed that students did make use of templates as building blocks in designing their plans. However, genuine understanding of templates were required in order to integrate the building blocks into programs.

Programming Learning Model

A programming learning model based on Linn's postulation was established using the path analyses. The main focus of the research was on the effects of language proficiency, template possession, and procedural skills on programming performance. The multiple regression analyses provided some

answers to the relative contributions to the variances in programming by the language proficiency, template possession and procedural skills scores. These analyses showed clearly the predominant contribution of procedural skills on programming, whether considered as separate entities or as an aggregate. The findings thus helped to clarify the role of language proficiency as antecedent to programming, as postulated by Linn (1985). While language proficiency was necessary, it was not sufficient, for problem solving in porgramming; and it underpinned programming developemnt through the mediating or facilitating effects of different levels of template abstraction and general sophistication with the procedural skills. This claim was based both on logical grounds and on the causal path analysis results.

Individual Differences On Programming

Factors hypothesized to influence outcomes from programming instruction in this research included general ability, gender, home computer possession and teacher-student ratio. This research revealed that students of high general ability performed better in all programming tasks. Correlations of ability and success in programming tasks ranged between .20 to .45.

Results in this research indicated that there were no gender differences in performance in template possession, testing skill, reformulating skill, planning skill and problem solving in programming. However, male students performed significantly better than their counterparts in language proficiency and reformulating skill. It was hypothesized that

male students outperformed in these two aspects was due to their more interest and experiences with computers.

Results in this research revealed that access to a home computer was not an important mediating factor to the performance of all programming tasks. Students did not possess home computer performed equally well in all aspects. It was attributed that students' access to computer equipments in schools were extensive, therefore possessing home computer was not beneficiary to performance in programming tasks.

In this research, teacher-student ratio was found to be an individual variable that related slightly to students programming performance. Students in low teacher-student ratio classes performed slightly better than students in high teacher-student ratio classes.

5.2 CONCLUSIONS

The main purpose of the present research is to study three aspects of programming. First, to investigate what componential knowledge and skills are related to programming. Second, to investigate how these knowledge and skills are affecting problem solving in programming. Third, to study what individual difference variables are significant to programming performance. The following conclusions were drawn in this research.

Results in this research indicated that knowledge including features of a programming language and templates abstracted from programming experiences; and procedural skills involving planning, testing and reformulating were significant components involved in programming.

Results from the protocol and causal path model analyses revealed that proficiency in the features of a programming language was a necessary but not a sufficient condition in problem solving in programming. Proficiency in the features of a programming language may underpin programming development through the mediating effects of different levels of template abstraction and the general sophistication with the procedural skills. Results from the regression and path model analyses indicated that procedural skills and especially planning skill was most prominent in problem solving in programming. The protocol and causal path model analyses revealed that template possession facilitated planning skill, therefore abstracting knowledge of template could bridge the gap between syntax learning and problem solving in programming.

In this research, ability was the individual difference variable that was most strongly and consistently related to student performance in all programming tasks. Gender differences among students were not strongly related to programming outcomes. Males and females did about equally well overall with males slightly outscoring females on some programming subtasks. Home computer possession was not an individual difference variable to programming performance. Students did not possess home computer performed equally well as those who possessed in all programming tasks investigated in this research. Teacher-student ratio related slightly to some programming outcomes. Students in classes with low teacher-student ratio slightly outperformed students in classes with high teacher-student ratio in some programming tasks.

5.3 LIMITATIONS

This research has limitations with respect to instrument development, research methodology, data analysis, and subject sampling. The four instruments for measuring language proficiency, template possession, procedural skills and problem solving ability in programming were newly developed by the researcher. Though a pilot study had been conducted and refinements had been made to improve the reliability of the instruments, further refinements were still needed.

Though clinical interviews had been conducted in the research to investigate programming behavior, the main focus of this research was a correlational study using paper and pencil tests to measure various programming tasks. Many details in the planning behavior, reformulating techniques, testing skills and problem solving process in programming might be overlooked or even could not be tapped by the written tests.

The results from the causal path model analyses could only be interpreted as a possible suggestion. It was analysed and interpreted in accordance with the programming learning model as postulated by Linn. However, this is not a unique way of interpretation. It can be explained in other ways. The procedural skills score was produced by adding the raw scores of planning, testing, and reformulating skills. Standardized score was not used for adding because of the limitation of the computational capability of the LISREL procedure in the SPSSX package. The measurement of the procedural skill might then be attenuated by the adding and the adding of raw scores.

The subjects in this research were not random sampled. They were selected at the convenience of the researcher due to the limitation of resources. Therefore, the generalization of the results of this research were limited.

5.4 RECOMMENDATIONS

Results from this research indicated that the gap between learning the features of a programming language and problem solving in programming can be bridged by learning templates and procedural skills. Procedural skills played a dominant role in problem solving in programming and it could be enhanced through template learning. Learning of both template and procedural skills based on the proficiency of a programming language. These findings have significant implications to various aspects of programming education and research area in this aspect.

First, programming curricular should not only mention the contents of the features of a programming language and program design, the commonly encountered programming tasks or algorithms required should also be listed. Various planning tools should also be incorportated and choice of usage should be allowed to encourage planning activities in programming.

Second, planning should be stressed in problem solving in programming. Appropriate tools should be taught to assist planning. Findings from these research indicated that preplanning was unlikely and flowcharting was an unfavourable planning tool. This is unfavourable to the betterment of more complicated program design and do not foster general problem solving skills. Two ways should be done to improve the status

quo. Problem inventory in programming courses should encourage planning. Result in this research indicated that the requirement of programming problems affect the planning behavior of students. Flowcharting as a planning tool should either be reconsidered or replaced. Very often, flowcharting was used as another form of program coding. The flow diagram becomes useless in planning when every minute details are represented in it. Flowcharting may be used as a planning tool only when the macro chunks of codes are represented as function boxes and critical control flow are actualized in the flow diagram. Template possession may facilitate this planning behavior. Other planning tools like stepwise refinement, structure diagram, psuedo code representation, and the like should be considered to replace or to supplement flowcharting as planning tools. Considerable reflection and research will be required to establish exactly how planning can be effectively taught and applied in programming design.

Third, practising teachers in teaching programming should not concentrate on teaching syntax of a programming language and then ask for program design. A spiral approach in teaching syntax, template, procedural skills and problem solving in programming should be adopted starting from easier problems to more complicated. Further promising research in studying the effect of the method of instruction which emphasizes explicit design of programs applying templates and procedural skills on programming performance is required.

The most potent individual difference variable found in this research related to programming performance was ability. High ability students outperformed low ability students by a

wide margin. However, the overall correlation between ability and programming performance was still not high indicating that programming depends not only on ability but also the knowledge and skills involved in programming. Therefore, method of instruction plays a significant role in learning programming. The results on studying the individual diference variables on home computer possession and teacher-student ratio also revealed that possessing home computers and increasing teacher-student ratio are not necessarily beneficiary to students' programming performance. If factors like effective programming pedagogy and teachers expertise in the content area are not addressed, additional resources like buying home computer or increase teacher-student ratio may only be a wastage. This research also indicated that male and female performed equally well in all programming tasks except the language proficiency and reformulating skill. This result indicated that there is no gender difference in the overall programming performance. The differences in the two tasks may be reduced or eliminated if girls are encouraged to gain more hand on experiences in learning the language features and debugging skills directly from the computer.

In sum, if programming education is perceived as a vehicle to foster problem solving skills, then the programming curriculum must be revised to include teaching template and procedural skills explicitly and treat them as important contents in programming courses.

BIBLIOGRAPHY

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. Memory & Cognition, 9(4), 422-433.

Allwood, C.M. (1986). Novices on the computer: A review of the literature. International Journal of Man-Machine Studies, 25, 633-658.

Anderson, J.R., Farrell, R. & Sauers, R. (1984). Learning to program in LISP. Cognitive Science, 8, 87-129.

Anderson, R.C. (1984). Some reflections on the acquisition of knowledge. Educational Researcher, 13, 5-10.

Atwood, M.E., & Ramsey, H.R. (1978). Cognitive structures in the comprehension and memory of computer programs: An investigation of computer debugging (Technical Report No. TR-78A21). Alexandria, VA: U.S. Army Reasearch Institute for the Behavioral and Social Sciences.

Atwood, M.E., Jeffries, R., & Polson, R.G. (1980). Studies in plan construction. I: Analysis of an extended protocol (Tech. Rep. No. SAI-80-028-DEN). Englewood, CO: Science Applications, Inc.

Ausubel, D.P. (1968). Educational psychology: A cognitive view. New York: Holt, Rinehart & Winston.

Bayman, P., & Mayer, R.E. (1983). A diagnosis of beginning programmers' misconceptions of BASIC programming statements. Communications of the ACM, 26(9), 677-679.

Becker, H.J., & Sterling, C.W. (1987). Equity in school computer use: National data and neglected considerations. Journal of Educational Computing Research, 3(3), 289-311.

Bishop, P. (1983). Computer programming in BASIC (2nd ed.). Hong Kong: Nelson.

Boulay, B.D. (1986). Some difficulties of learning to program. Journal of Educational Computing Research, 2(1), 57-73.

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. International Journal of Man-Machine Studies, 9, 737-751.

Bushke, H. (1976). Learning is organized by chunking. Journal of Verbal Learning and Verbal Behavior, 15, 313-324.

Chaudhary, B.D. (1985). A study in dimensions of psychological complexity of programs. International Journal of Man-Machine Studies, 23, 113-133.

Clements, C.A., Kurland, D.M., Mawby, R. & Pea, R.D. (1986). Analogical reasoning and computer programming. Journal of Educational Computing Researech, 2(4), 473-485.

Curriculum Development Committee (1986). Syllabus for computer studies (Form IV - V). Hong Kong.

Dahl, O.J., Dijkstra, E.W., & Hoare, C.A.R. (1973). Structured programming. London-New York:Academic Press.

Dalbey, J., & Linn, M.C. (1985). The demands and requirements of computer programming: A literature review. Journal of Educational Computing Research, 1(3), 253-274.

Dalbey, J., & Linn, M.C. (1986). Cognitive consequences of programming: Augmentations to BASIC instruction. Journal of Educational Computing Research, 2(1), 75-93.

Dalbey, J., Tourniaire, F., & Linn, M.C. (1986). Making programming instruction cognitively demanding: An intervention study. Journal of Research in Science Teaching, 23(5), 427-436.

Enrlich, K., & Soloway, E. (1984). An empirical investigation of the tacit plan knowledge in programming. In J. Thomas, & M. Schneider (Eds.), Human factors in computer systems. Norwood, NJ: Ablex.

Findlay, W., & Watt, D.A. (1981). PASCAL: An introduction to methodical programming (2nd ed.). Massachusetts: Pitman.

Fisher, C., & Mandinach, E. (1985, March). Individual differences and acquisition of computer programming skill. Paper presented at the annual meeting of the American Educational Research Association, Chicago.

Gagne, R.M. (1985). The conditions of learning and theory of instruction (4th ed.). NY: CBS College.

Galanter, E. (1983). Kids and computers: The parents' microcomputer handbook. NY: Putnam.

Gick, M.L. (1986). Problem-solving strategies. Educational Psychologist, 21(1 & 2), 99-120.

Gick, M.L., & Holyoak, K.J. (1983). Schema induction and analogical transfer. Cognitive Psychology, 15, 1-39.

Glaser, R. (1984). Education and thinking: The role of knowledge. American Psychology. 39(2), 93-104.

Hawkins, J. (1987). Computers and girls: Rethinking the issues. In R.D. Pea, & K. Sheingold (Ed), Mirrors of minds: Patterns of experience in educational computing. Norwood NJ: Ablex.

Hayes-Roth, B., & Hayes-Roth, F. (1979). A cognitive model of planning. Cognitive Science, 3, 275-310.

IFIP working conference on the role of programming in teaching informatics. (1984).

Jeffries, R., Turner, A.A., Polson, P.G., & Atwood, M.E. (1981). The processes involved in designing software. In J.R. Anderson (Ed.), Cognitive skills and their acquisition. Hillsdale, NJ: Erlbaum.

Johanson, R.P. (1988). Computers, cognition and curriculum: retrospect and prospect. Journal of Educational Computing Research, 4(1), 1-30.

Johnson, W.L., & Soloway, E. (1983). PROUST: Knowledge-based program understanding. Proceedings of the 7th international conference on software engineering IEEE, Orlando, Florida.

Kintsch, W., & Greeno, J. (1985). Understanding and solving word arithmetic problems. Psychological Review, 92, 109-129.

Kreitzberg, B., & Swanson, L. (1974). A cognitive model for structuring an introductory programming curriculum. AFIPS Proceedings of the National Computer Conference. Montvale, NJ: AFIPS.

Kurland, D.M., Clement, K., Mawby, R. & Pea, R.D. (1984). Mapping the cognitive demands of learning to program (Technical Reprot No. 29). NY: Bank Street College of Education, Center for Children and Technology.

Kurland, D.M., Mawby, R., & Cahir, N. (1984, April). The development of programming expertise in adults and children. Paper presented at the annual meeting of the American Education Research Association, New Orleans, LA.

Kurland, D.M., Pea, R.D., Clement, C. & Mawby, R. (1986). A study of the development of programming ability and thinking skills in high school students. Journal of Educational Computing Research, 2(4), 429-458.

Li, N.Y. (1986). A study of the norms of Hong Kong students of age groups 15 to 18 in the performance of Raven's advanced progressive matrices test. M.A.(ED) Thesis, The Chinese University of Hong Kong.

Linn, M.C. (1985). The cognitive consequences of programming instruction in classrooms. Educational Researcher, 14, 9-16.

Linn, M.C., & Dalbey, J. (1985). Cognitive consequences of programming instruction: Instruction, Access, and Ability. Educational Psychologist, 20(4), 191-206.

Linn, M.C., Sloane, K.D., & Clancy, M.J. (1987). Ideal and actual outcomes from precollege PASCAL instruction. Journal of research in science teaching, 24(5), 467-490.

Lo, L.F. (1982). Testing the aptness of the multiple regression model. CUHK Educational Journal, 10(2).

Mandinach, E., & Linn, M.C. (1986). The cognitive effects of computer learning environments. Journal of Educational Computing Research, 2(4), 411-427.

Mandinach, E., Linn, M.C., Pea, R.D. & Kurland, D.M. (1987). Cognitive consequences of programming: Achievements of experienced and talented students. Journal of Educational Computing Research, 3(1), 53-72.

Mayer, R.E. (1979). A psychology of learning BASIC. Communications of the ACM, 22(11), 589-593.

Mayer, R.E. (1981). The psychology of how novices learn computer programming. Computing Surveys, 13(1), 121-141.

Mayer, R.E. (1985). Learning in complex domains: A cognitive analysis of computer programming. The Psychology of Learning and Motivation, 19, 89-130.

McDonald, J.L. (1987). Subroutine supersleuths. The Computing Teacher, May, 24-26.

McKeithen, K.B., Reitman, J.S., Rueter, H.H. & Hirtle, S.C. (1981). Knowledge organization and skill differences in computer programmers. Cognitive psychology, 13, 307-325.

Newell, A., & Simon, H. (1972). Human problem solving. Englewood Cliffs, NJ: Prentice Hall.

Nie, N.H. (Ed.). (1983). SPSSX. N.Y.: McGrawHill.

Norman, D., Gentner, D., & Stevens, A. (1976). Comments on learning schemata and memory representation. In D. Klahr (Ed.), Cognitive and instruction. Hillsdale, NJ: Erlbaum.

Nutter, J.T., & Hassell, J. (1985). Programming schemes in program understanding and maintenance: An introduction. AEDS Journal, 18(3), 195-206.

Papert, S. (1980). Mindstorms: Children, computers & powerful ideas, Basics Books, NY.

Pea, R.D. (1983, April). Logo programming and problem solving. Paper presented at the annual meeting of the American Educational Research Association, Montreal.

Pea, R.D. (1986). Language-independent conceptual "bugs" in novice programming. Journal of Educational Computing Research, 2(1), 25-35.

Pea, R.D., Hawkins, J., Clement, C.A., & Mawby, R. (1984). The development of expertise in Logo by children. NY: Bank Street College of Education, Center for Children and Technology.

Pea, R.D., & Kurland D.M. (1984 March). Logo programming and the development of planning skills (Technical Report No. 16). NY: Bank Street College of Education, Center for Children and Technology.

Pea, R.D., & Kurland, D.M. (1987). On the cognitive effects of learning computer programming. In R.D. Pea, & K. Sheingold (Eds.), Mirrors of minds: Patterns of experience in educational computing. Norwood, NJ: Ablex.

Pennington, N. (1982). Cognitive components of expertise in computer programming: A Review of the Literature, (Technical Report No. 46). University of Michigan, Center for Cognitive Science, Ann Arbor.

Perkins, D.N., Hancock, C., Hobbs, R., Martin, F. & Simmons, R. (1986). Conditions of learning in novice programmers. Journal of Educational Computing Research, 2(1), 37-55.

Pintrich, P.R., Berger, C.F., & Stemmer, P.M. (1987). Students' programming behavior in a PASCAL course. Journal of Research in Science Teaching, 24(5), 451-466.

Polya, G. (1957). How to solve it: A new aspect of mathematical method (2nd ed.). Princeton, NJ: Princeton.

Putnam, R.T., Sleeman, D., Baxter, J.A., & Kuspa, L.K. (1986). A summary of misconceptions of high school BASIC programmers. Journal of Educational Computing Research, 2(4), 459-472.

Raven, J.C. (1958). Advanced Progressive Matrices. NY: Psychological Corporation.

Ratcliff, B., & Siddiqi, J.I.A. (1985). An empirical investigation into problem decomposition strategies used in program design. International Journal of Man-Machine Studies, 22, 77-90.

Reed, S.K., Ernst, G.W. & Banerji, R. (1974). The role of analogy in transfer between similar problem states. Cognitive Psychology, 6, 436-450.

Rumelhart, D., & Norman, D. (1981). Analogical processes in learning. In J.R. Anderson (Ed.), Cognitive skills and their acquisition, 335-359.

Salomon, G., & Perkins, D.N. (1987). Transfer of cognitive skills from programming: When and how ? Journal of Educational Computing Research, 3(2), 149-169.

Scandura, J.M. (1977). Problem solving: A structural/process approach with instructional implications. NY: Academic Press.

Shin, T.F. (1987). A review on computing education. Paper presented in the conference on microcomputers in education, Hong Kong.

Shneiderman, B. (1976a). A review of design techniques for program & data, Software: Practice and Experience, 6, 555-567.

Shneiderman, B. (1976b). Exploratory experiments in programmer behavior. International Journal of Computer and Information Sciences, 5(2), 123-143.

Shneiderman, B. (1977a). Measuring computer program quality and comprehension. International Journal of Man-Machine Studies, 9, 465-478.

Shneiderman, B. (1977b). Teaching programming: A spiral approach to syntax and semantics. Computing & Education, 1, 193-197.

Shneiderman, B. (1980). Software psychology. Cambridge, Massachusetts: Winthrop.

Shneiderman B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. International Journal of Computer and Information Sciences, 8(3), 219-238.

Shneiderman, B., Mayer, R., Mckay, D., & Heller, P. (1977). Experimental investigations of the utility of detailed flowcharts in programming. Communications of the ACM, 20(6), 373-381.

Simon, H.A. (1973). The structure of ill-structured problems. Artificial Intelligence, 4, 181-201.

Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive strategies & looping construct: An empirical study. Communications of the ACM, 26(11), 853-860.

Soloway, E., & Enrlich, K. (1984). Empirical studies of programming knowledge. IEEE Transactions on Software Engineering, SE-10(5), 595-609.

Thorndyke, P.W., & Hayes-Roth, B. (1979). The use of schemata in the acquisition and transfer of knowledge. Cognitive Psychology, 11, 82-106.

Webb, N.M. (1985). Cognitive requirements of learning computer programming in group and individual settings. AEDS Journal, 18(3), 183-193.

Webb, N.M. (1984). Microcomputer learning in small groups: cognitive requirements and group processes. Journal of Educational Psychology, 76, 1076-1088.

Webb, N.M., Ender, P. & Lewis, S. (1986). Problem-solving strategies and group processes in small groups learning computer programming. American Educational Research Journal, 23(2), 243-261.

Webb, N.M. & Shavelson, R.J. (1985). Computers, education, and educational psychologists. Educational Psychologist, 20(4), 163-185.

Weinberg, G.M. (1970). The psychology of computer programming. NY: Van Nostrang Reinhold.

Weiser, M. (1982). Programmers use slicers when debugging. Communciations of the ACM, 25.

Wirth, N. (1971). Program development by stepwise refinement. Communications of the ACM, 14(4), 221-227.

APPENDIX A

(1) Language Proficiency Test

INSTRUCTIONS

1. Write down your name, class and class number and personal particulars on the answer sheet provided.

2. There are 18 multiple choice questions in this test. Answer ALL questions. You have 30 minutes.

3. You may refer to the following information whenever necessary.

A Partial Character list for ASCII

Character	ASCII	Character	ASCII
0	48	F	70
1	49	G	71
2	50	H	72
3	51	I	73
4	52	J	74
5	53	K	75
6	54	L	76
7	55	M	77
8	56	N	78
9	57	O	79
:	58	P	80
;	59	Q	81
<	60	R	82
=	61	S	83
>	62	T	84
?	63	U	85
@	64	V	86
A	65	W	87
B	66	X	88
C	67	Y	89
D	68	Z	90
E	69		

List of operators and Reserved Words

+, -, *, /, ^

>, <, =, >=, <=, <>

AND, OR, NOT

SQR, INT, RND, ABS, SIN, COS, TAN, ATN, LOG, EXP

+(string concatenation), LEN, ASC, VAL, STR$, LEFT$, MID$, RIGHT$, STRING$

PRINT, PRINT TAB, INPUT, DATA, READ, RESTORE, PRINT USING

DIM, STOP, END, GOTO, GOSUB, RETURN, FOR ... TO ... STEP, NEXT, REM, IF ... THEN, IF ... THEN ... ELSE, ON ... GOSUB, ON ... GOTO, ON ERROR

CLS, HOME

1. Which of the following pairs of variable names and given data match each other in BASIC ?

	Variable	Data
(1)	P1	6.543
(2)	Q2$	6543
(3)	R3%	6,543

A. (2) only
B. (3) only
C. (1) and (2) only
D. (1) and (3) only
E. (1), (2) and (3)

2.
```
10 A$ = "I"
20 I = 3
30 A$ = A$ + "***"
40 PRINT I+3-1;A$;"A$"
50 END
```

When the above program is executed, the output is
A. 5***A$
B. 53***A$
C. 5I***A$
D. I+3-1I***A$
E. 3+3-1A$I

3.
```
10 PRINT TAB(1);"1234567890"
20 K = 2
30 PRINT TAB(K);"*";TAB(K+3);"*";
40 PRINT
50 PRINT TAB(2*K);"*";TAB(K+4)"*"
60 END
```

When the above program is executed, the output is

A.
```
1234567890
 *  *
   * *
```

B.
```
1234567890
 *    *
   *    *
```

C.
```
1234567890
 * ***
```

D.
```
1234567890
 *  *
    * *
```

E.
```
1234567890
 *    *
      *    *
```

4. Which of the following INPUT statement(s) will ACCEPT those data in brackets WITHOUT SYNTAX ERROR OR REENTER ?

(1)	10 INPUT "BASE ";B,"HEIGHT ";H	(2, 3)
(2)	20 INPUT "YOUR NAME ";N$	(N$)
(3)	30 INPUT "(Y FOR YES) ";X	(Y)

A. (1) only
B. (2) only
C. (3) only
D. (2) and (3) only
E. (1), (2) and (3)

5.
```
10 DIM A(9,9), B(9)
20 FOR J = 1 TO 2
30    FOR I = 1 TO 4
40       INPUT A(J,I)
50       A(J,I) = A(J,I) + J
60    NEXT I
70 NEXT J
80 B(1) = A(1,2) + A(2,2)
90 PRINT B(1)
100 END
```

When the above program is executed, 0,9,1,2,4,8,6,5 are inputted sequentially from keyboard, the output is

A. 6
B. 14
C. 16
D. 20
E. SYNTAX ERROR IN LINE 80

6.
```
10 RESTORE
15 READ EVEN$
20 READ ODD$
30 DATA 3,4
40 READ A3$, A4$
50 DATA FOUR
60 DATA THREE
70 PRINT A3$,EVEN$,A4$,ODD$
80 END
```

When the above program is executed, the output is

A. 3	4	FOUR	THREE
B. THREE	4	FOUR	3
C. FOUR	4	THREE	3
D. 3	FOUR	4	THREE
E. FOUR	3	THREE	4

7.
```
10 Q = 0
20 READ P
30 IF P = 0 THEN GOTO 70
40 IF NOT(P >= 0 AND P <= 5) THEN GOTO 60
50 Q = Q + P
60 GOTO 20
70 PRINT Q
80 DATA 1, -1, 2, 5, -6, 10, -2, 0
90 END
```

When the above program is executed, the output is
A. 1
B. 3
C. 8
D. 9
E. 18

8.
```
10 Y$ = ""
20 FOR I = 1 TO 5
30    READ X$
40    IF X$ <= STR$(I) THEN Y$ = Y$ + X$
50    Y$ = X$ + Y$
60 NEXT I
70 Y$ = Y$ + STR$(I)
80 PRINT Y$
90 DATA 8,1,5,4,3
100 END
```

When the above program is executed, the output is

A. 345181435
B. 345181436
C. 581435
D. 581436
E. 341815435

9.
```
10 READ FIRST, SECOND
20 IF FIRST = SECOND THEN GOTO 80
30 IF FIRST > SECOND THEN FIRST = FIRST - SECOND : GOTO 50
40 SECOND = SECOND - FIRST
50 PRINT FIRST, SECOND,
60 GOTO 10
70 DATA 2, 1, 5, 7, 7, 7
80 END
```

When the above program is executed, the output is

A. 1 1 2 5
B. -1 2 2 5
C. 1 0 2 5
D. 1 1 5 2
E. 1 1 5 2 7 7

10.
```
10 L = 22
20 IF L >= 40 AND L <= 60 THEN GOTO 60
30    IF L < 40 THEN L = L + 50
40    IF L > 60 THEN L = L - 10
50 GOTO 20
60 PRINT L
70 END
```

When the program on the left is executed, the output is

A. 52
B. 12
C. 62
D. 22
E. 72

11.
```
10 FOR I = 1 TO 3
20    GOSUB 60
30    PRINT S,
40 NEXT I
50 REM  STOP
60 S= 0
70 FOR J = 1 TO I
80    S = S + J
90 NEXT J
100 RETURN
110 END
```

When the program on the left is executed, the output is

A. 1 3 6
B. 1 3 6
   SYNTAX ERROR IN LINE 100
C. 1 4 10
D. 1 1 1
E. 1 3 6
   SYNTAX ERROR IN LINE 50

12.
```
10 M = 2
20 GOSUB 50
30 PRINT M
40 GOTO 100
50 M = M + 1
60 GOSUB 80
70 RETURN
80 M = (M + 1)*(M-1)^2
90 RETURN
100 END
```

When the program on the left is exectued, the output is

A. 4
B. 3
C. 2
D. 17
E. 16

13.
```
10 A = 4
20 B = 2
30 A = A * B
40 B = A / B
50 A = A / B
60 PRINT A, B
70 END
```

When the above program is executed, the output is

A. 8 2
B. 8 4
C. 1 4
D. 4 2
E. 2 4

14. In a machine, RND(1) will generate a random number greater than or equal to 0 but less than 1.

```
10 X = RND(1) * 12
20 Y = INT(X)
30 Z = ABS(Y - 12)
40 PRINT Z
50 END
```

When the above program is exectued in the machine, the output may be an integer from

A. 0 to 11
B. 0 to 12
C. 1 to 12
D. -12 to -1
E. -12 to 0

15.
```
10 A$ = "0123456789"
20 B = VAL(LEFT$(A$,2))
30 C = VAL(MID$(A$,5,2))
40 D = VAL(RIGHT$(A$,2))
50 E = B + C
60 PRINT STR$(E) + STR$(D)
70 END
```

When the above program is executed, the output is

A. 135
B. 1341
C. 4698
D. 5789
E. 4689

16.
```
10 A$ = "A BC DEF"
20 B = LEN(A$)
30 C$ = LEFT$(A$, B-2)
40 C$ = RIGHT$(C$, 3)
50 PRINT C$
60 END
```

When the above program is executed, the output is

A. BCD
B. BC
C. BC
D. C D
E. D C

17.
```
10 P = 6
20 Q = 4
30 R = 2
40 S = P + Q / R
50 T = SQR(S - 4)/ 2 * R
60 PRINT T
70 END
```

When the above program is executed, the output is

A. 4
B. 2
C. 1
D. 0.5
E. 0.25

18. REM statements in BASIC programs are used for

(1) describing the function of the program lines.
(2) helping the computer to interpret the programs.
(3) execution as the other BASIC statements such as GOTO, PRINT, INPUT etc.

A. (1) only
B. (2) only
C. (1) and (2) only
D. (2) and (3) only
E. (1), (2), amd (3)

**** END OF THIS TEST ****

APPENDIX B

(2) TEMPLATE MEASUREMENT TEST

INSTRUCTIONS

1. Write down your name, class, and class number on the answer sheet provided.

2. There are 18 multiple choice questions in this test. Answer ALL questions.

3. You have 35 minutes.

4. You may refer to the following information whenever necessary.

List of Operators and Reserved Words

+, -, *, /, ^

>, <, =, >=, <=, <>

SQR, INT, RND, ABS, SIN, COS, TAN, ATN, LOG, EXP

+(string concatenation), LEN, ASC, VAL, STR$, LEFT$, MID$, RIGHT$, STRING$

PRINT, PRINT TAB, INPUT, DATA, READ, RESTORE, PRINT USING

DIM, STOP, END, GOTO, GOSUB, RETURN, FOR ... TO ...STEP, NEXT, REM, IF ... THEN, IF ... THEN ... ELSE, ON ... GOSUB, ON ... GOTO, ON ERROR

CLS, HOME

1.
```
10 _____
20 INPUT X
30 IF X = 9999 THEN GOTO 60
40 _____
50 GOTO 20
60 PRINT " THERE ARE ";A;" NUMBERS."
70 END
```

In order to COUNT the number of inputted values excluding 9999, line 10 and 40 in the above program should BEST be

A.
```
10 INPUT A
40 A = A + X
```

B.
```
10 A = 0
40 A = A + X
```

C.
```
10 INPUT A
40 A = A + 1
```

D.
```
10 X = 1
40 A = A + X
```

E.
```
10 A = 0
40 A = A + 1
```

2. The assignment statement to round off a real number to 2 decimal places should be

A. Y = INT(X*100)/100 + 0.5

B. Y = INT(X*100 + 0.5)/100

C. Y = INT(X + 0.5)/100

D. Y = INT(X*1000 + 0.5)/1000

E. Y = INT((X + 0.5)*100 + 0.5)/100

3.
```
10 IF A$ > B$ THEN PRINT A$ : GOTO 30
20 PRINT B$
30 ...
```

The program segment is to

A. compare A$ and B$.

B. print contents in A$ and continuous with line 30.

C. print contents in B$ and continuous with line 30.

D. select contents from A$ or B$ for printing.

E. skip line 20.

4.
```
100 INPUT N
200 IF (N <= 0) OR (N <> INT(N))  THEN GOTO 100
300 ...
```

The purpose of the program segment is to ensure that the inputs are

A. positive numeric values or integral numbers.
B. positive non-integral numeric values.
C. positive integers.
D. positive integers or zero.
E. negative non-integral numeric values or zero.

5 .
```
10 _________________
20 INPUT N
30 FOR I = 1 TO N
40    INPUT X
50 _________________
60 NEXT I
70 PRINT "THE SUM = ";A
80 END
```

In order to sum N data inputted from keyboard, lines 10 and 50 should BEST be

A.
```
10 INPUT A
50 A = A + X
```

B.
```
10 A = 0
50 A = A + I
```

C.
```
10 A = 0
50 A = A + X
```

D.
```
10 INPUT A
50 A = A + N
```

E.
```
10 A = 0
50 A = A + 1
```

6. Which of the following program segment allow only "Y" , "N" to be accepted ?

A.
```
10 INPUT "CONTINUE (Y/N) ?";YN$
20 IF YN$ = "Y" OR YN$ = "N" THEN GOTO 10
30 ...
```

B.
```
10 INPUT "CONTINUE (Y/N) ?";YN$
20 IF (YN$ <> "Y") AND (YN$ <> "N") THEN GOTO 10
30 ...
```

C.
```
10 INPUT "CONTINUE (Y/N) ?";YN$
20 IF (YN$ <> "Y") OR (YN$ <> "N") THEN GOTO 10
30 ...
```

D.
```
10 INPUT "CONTINUE (Y/N) ?";YN$
20 IF YN$ <> "Y" THEN GOTO 10
30 IF YN$ <> "N" THEN GOTO 10
40 ...
```

E.
```
10 INPUT "CONTINUE (Y/N) ?";YN$
20 IF NOT (YN$ = "Y" AND YN$ = "N") THEN GOTO 10
30 ...
```

7.
```
***
***
***
***
```

Which of the following program segment will produce the pattern of output as shown in the left side ?

A.
```
10 FOR I = 1 TO 4
20 FOR J = 1 TO 3
30    PRINT "*"
40 NEXT J
50 NEXT I
```

B.
```
10 FOR I = 1 TO 4
20 FOR J = 1 TO 3
30    PRINT "*";
40 NEXT J
50 NEXT I
```

C.
```
10 FOR I = 1 TO 3
20 FOR J = 1 TO 4
30    PRINT "*";
40 NEXT J
50 NEXT I
```

D.
```
10 FOR I = 1 TO 4
20 FOR J = 1 TO 3
30    PRINT "*";
40 NEXT J
50 PRINT
60 NEXT I
```

E.
```
10 FOR I = 1 TO 3
20 FOR J = 1 TO 4
30    PRINT "*";
40 NEXT J
50 PRINT
60 NEXT I
```

8.
```
10 M = A(1)
20 FOR I = 2 TO N
30    IF A(I) >= M THEN M = A(I)
40 NEXT I
50 PRINT M
60 END
```

The above program is to find

A. the largest value among A(1) to A(N).
B. the smallest value among A(1) to A(N).
C. a value equal to M among A(2) to A(N).
D. all values greater than or equal to M from A(2) to A(N).
E. a value greater than or equal to A(1) from A(2) to A(N).

9.
```
10 INPUT F$
20 FOR I = 1 TO N
30    _____________________________
40 NEXT I
50 _____________________
60 ....
```

Array A$ have unequal strings in it. In order to find F$ in the array A$, the blank lines should BEST be

A.
```
30 IF F$ <> A$(I) THEN GOTO 40
50 PRINT A$(I)
```

B.
```
30 IF F$ = A$(I) THEN PRINT I
50 PRINT "FOUND"
```

C.
```
30 IF F$ = A$(I) THEN PRINT I : GOTO 60
50 PRINT "FOUND"
```

D.
```
30 IF F$ = A$(I) THEN PRINT I
50 PRINT "NOT FOUND"
```

E.
```
30 IF F$ = A$(I) THEN PRINT I : GOTO 60
50 PRINT "NOT FOUND"
```

10.
```
10 _____________
20 INPUT X
30 IF X = -999 THEN GOTO 60
40 ____________________________
50 GOTO 20
60 PRINT "SMALLEST VALUE : ";M
70 END
```

In order to find the smallest value among the inputted positive or negative values except -999, the blank lines should BEST be

A.
```
10 INPUT M
40 IF X < M THEN X = M
```

B.
```
10 M = 0
40 IF X < M THEN M = X
```

C.
```
10 INPUT M
40 IF X > M THEN M = X
```

D.
```
10 INPUT M
40 IF X < M THEN M = X
```

E.
```
10 M = 0
40 IF X > M THEN M = X
```

11. Which of the following program segment can be used to sum an array A with N elements ?

A.
```
10 C = 0
20 FOR I = 1 TO N
30     C = C + I
40 NEXT I
```

B.
```
10 C = 0
20 FOR I = 1 TO N
30     C = C + A(I)
40 NEXT I
```

C.
```
10 C = 0
20 FOR I = 1 TO N
30     C = C + 1
40 NEXT I
```

D.
```
10 C = 1
20 FOR I = 2 TO N
30     C = C + A(I)
40 NEXT I
```

E.
```
10 C = 0
20 FOR I = 1 TO N
30     C = C + A
40 NEXT I
```

12. Which of the following CANNOT be used to read data into an array A with more than 1 element ?

A.
```
10 N = 0
20 PRINT N
30 FOR I = 1 TO N
40    READ A(I)
50 NEXT I
60 ...
```

B.
```
10 N = 999
20 FOR I = 1 TO N
30    READ A(I)
40    IF A(I) = 999 THEN GOTO 50
45 NEXT I
50 ...
```

C.
```
10 READ N
20 FOR I = 1 TO N
30    READ A(I)
40 NEXT I
50 ...
```

D.
```
10 INPUT N
20 FOR I = 1 TO N
30    READ A(I)
40 NEXT I
50 ...
```

E.
```
10 I = 1
20 READ A(I)
30 IF A(I) = 0 THEN GOTO 60
40 I = I + 1
50 GOTO 20
60 ...
```

13. A student writes a program to PRINT PRIME NUMBERS from 3 to M.

```
10 FOR N = 3 TO M
20 _____(a)_____
30 FOR I = 2 TO N -1
40    IF (N - I*INT(N/I) = 0) THEN ____(b)____
50 NEXT I
60 _____(c)_____
70 NEXT N
```

Fill in the blank lines to complete the program segment.

A.
```
(a) P = 1
(b) P = 0
(c) IF P = 0 THEN PRINT N
```

B.
```
(a) P = 0
(b) P = 1
(c) IF P = 1 THEN PRINT N
```

C.
```
(a) P = 1
(b) P = 0
(c) IF P = 1 THEN PRINT N
```

D.
```
(a) P = 1
(b) P = 0
(c) IF P = 1 THEN PRINT I
```

E.
```
(a) P = 0
(b) P = 1
(c) IF P = 1 THEN PRINT M
```

Questions 14 to 16 refer to the program segment below.

The program segment is a BUBBLE SORT algorithm.

```
10  FOR J = 1 TO N - 1
20  FOR I = 1 TO N - 1
30  IF A(I) < A(I+1) THEN GOTO 70
40  _____________
50  _____________
60  _____________
70  NEXT I
80  NEXT J
90  ...
```

14. Complete the program segment by filling in the blank lines.

A.
```
40 A(I) = T
50 A(I + 1) = A(I)
60 T = A(I + 1)
```

B.
```
40 T = A(I + 1)
50 A(I) = A(I + 1)
60 A(I + 1) = T
```

C.
```
40 T = A(I)
50 A(I + 1) = A(I)
60 A(I) = T
```

D.
```
40 T = A(I)
50 A(I) = A(I + 1)
60 A(I + 1) = T
```

E.
```
40 A(I) = T
50 A(I + 1) = A(I)
60 T = A(I + 1)
```

15. Which of the following CHANGES in line 20 and 30 will allow the most efficient way to sort the array A in ASCENDING order ?

A.
```
20 FOR I = 1 TO N - 1
30 IF A(I) >= A(I + 1) THEN GOTO 70
```

B.
```
20 FOR I = 1 TO N - J + 1
30 IF A(I) > A(I + 1) THEN GOTO 70
```

C.
```
20 FOR I = 1 TO N - J
30 IF A(I) <= A(I + 1) THEN GOTO 70
```

D.
```
20 FOR I = 1 TO N - J
30 IF A(I) >= A(I + 1) THEN GOTO 70
```

E.
```
20 FOR I = 1 TO N - J + 1
30 IF A(I) < A(I + 1 ) THEN GOTO 70
```

16. By adding 3 lines in the program segment, the number of passes may be reduced. Which of the following insertions can work in the BEST way ?

A.
```
15 S = 0
65 S = 1
75 IF S = 0 THEN GOTO 90
```

B.
```
15 S = 1
65 S = 0
75 IF S = 0 THEN GOTO 90
```

C.
```
 5 S = 0
65 S = 1
75 IF S = 0 THEN GOTO 90
```

D.
```
25 S = 0
65 S = 1
75 IF S = 0 THEN GOTO 90
```

E.
```
15 S = 0
75 S = 1
77 IF S = 1 THEN GOTO 90
```

17. The following is a BINARY SEARCH subroutine to search V in array A which have N elements.

```
10 L = 1
20 R = N
30 M = INT((L + R)/2)
40 IF (V = A(M)) OR (L > R) THEN GOTO 80
50 IF (V < A(M)) THEN _____(a)_____
60 IF (V > A(M)) THEN _____(b)_____
70 GOTO 30
80 IF (L > R) THEN _____(c)_____
90 RETURN
```

If the contents in array A are sorted in ASCENDING order, the blank lines should be

A.
```
(a) R = M + 1
(b) L = M - 1
(c) PRINT "NOT FOUND"
```

B.
```
(a) R = M + 1
(b) L = M - 1
(c) PRINT "FOUND"
```

C.
```
(a) R = R - 1
(b) L = L + 1
(c) PRINT "NOT FOUND"
```

D.
```
(a) R = M - 1
(b) L = M + 1
(c) PRINT "FOUND"
```

E.
```
(a) R = M - 1
(b) L = M + 1
(c) PRINT "NOT FOUND"
```

18. The following is a program segment to FIND the H.C.F. of a set of positive integers store in A(1) to A(N). Fill in the blanks with lines of BASIC code which in your opinion best completes the program.

```
10 M = A(1)
20 FOR I = 2 TO N
30    IF A(I) < M THEN M = A(I)
40 NEXT I
50 _____(a)_____
60 FOR J = 2 TO M
70    _____(b)_____
80    FOR I = 1 TO N
90       IF A(I)/J <> INT(A(I)/J) THEN _____(C)_____
100   NEXT I
110   IF F = 0 THEN GOTO 160
120   HCF = HCF * J
130   FOR I = 1 TO N
140      A(I) = A(I)/J
150   NEXT I
160 NEXT J
170 PRINT " THE H.C.F. IS ";HCF
180 ...
```

A.
```
(a)  HCF = 1
(b)  F = 1
(C)  F = 0
```

B.
```
(a)  F = 1
(b)  HCF = 1
(c)  F = 0
```

C.
```
(a)  HCF = 0
(b)  F = 1
(c)  F = 0
```

D.
```
(a) HCF = 1
(b) F = 0
(c) F = 1
```

E.
```
(a) F = 0
(b) HCF = 1
(c) F = 1
```

END OF THIS TEST

APPENDIX C

(3) TEST ON PROCEDURAL SKILLS

INSTRUCTIONS

1. Write down your name, class and class number on the answer sheet provided.

2. There are 8 questions in this test. Answer ALL questions. You have 35 minutes.

3. You may refer to the following information whenever necessary.

List of operators and Reserved Words

+, -, *, /, ^

>, <, =, >=, <=, <>

AND, OR, NOT

SQR, INT, RND, ABS, SIN, COS, TAN, ATN, LOG, EXP

+(string concatenation), LEN, ASC, VAL, STR$, LEFT$, MID$, RIGHT$, STRING$

PRINT, PRINT TAB, INPUT, DATA, READ, RESTORE, PRINT USING

DIM, STOP, END, GOTO, GOSUB, RETURN, FOR ... TO ... STEP, NEXT, REM, IF ... THEN, IF ... THEN ... ELSE, ON ... GOSUB, ON ... GOTO, ON ERROR

CLS, HOME

Sample question

1. Assume that the following program is correct and that the inputs are all integers.

```
10 INPUT A,B
20 IF A > B THEN PRINT A;">";B : GOTO 50
30 IF A < B THEN PRINT B;">";A : GOTO 50
40 PRINT A;"=";B
50 END
```

Design no more than 3 sets of test data to FULLY test the above program in responding to the INPUT statement in line 10.

Answer

1. (i) 3,4
   (ii) 4,2
   (iii) 1,1

1. Assume that the following program is correct and that the inputs are real numbers.

```
10 PI = 3.14159
20 INPUT "RADIUS OF CIRCLE  ";R
30 IF R = 999 THEN GOTO 90
40 IF R < 0 THEN PRINT "IMAGINARY CIRCLE" : GOTO 80
50 IF R = 0 THEN PRINT "POINT CIRCLE" : GOTO 80
60 A = PI * R * R
70 PRINT "REAL CIRCLE WITH AREA OF CIRCLE = ";A
80 GOTO 20
90 END
```

Design no more than 4 sets of test data to FULLY test the above program in responding to the INPUT statement in line 20.

2. Assume that the following program is correct and that all data in line 130 are integers.

```
10 FAIL = 0
20 PASS = 0
30 CREDIT = 0
40 READ MK
50 IF MK = 0 THEN GOTO 100
60 IF MK >= 1  AND MK < 50 THEN FAIL = FAIL + 1
70 IF MK >= 50 AND MK <= 99 THEN PASS = PASS + 1
80 IF MK >= 80 AND MK <= 99 THEN CREDIT = CREDIT + 1
90 GOTO 40
100 PRINT "FAILING = ";FAIL
110 PRINT "PASSING = ";PASS
120 PRINT "CREDITING = ";CREDIT
130 DATA _____________________________
140 END
```

Design 1 set of test data for line 130 with no more than 8 items of data to FULLY test the above program.

3. The following program is used for COUNTING THE NUMBER OF WORDS in a string A$. The string contains of WORDS in alphabets A to Z and/or SPACE(S) only.

```
10 BK$ = " "
20 C = 0 : W$ = ""
30 INPUT A$
40 A$ = A$ + BK$
50 FOR I = 1 TO LEN(A$)
60    B$ = MID$(A$,I,1)
70    IF B$=BK$ AND LEN(W$) <> 0 THEN C=C+1 : W$="" : GOTO 100
80    IF B$=BK$ AND LEN(W$) = 0 THEN GOTO 100
90    W$ = W$ + B$
100 NEXT I
110 PRINT C
120 END
```

Design no more than 4 sets of test data to FULLY test the above program in responding to the INPUT statement in line 30.

Give each set of data inside QUOTATION MARKS.

4. A student writes the following program to translate Arabic number (1 to 10) to English word. Assume that it is correct.

```
10 DIM E$(10)
20 FOR I = 1 TO 10
30     READ E$(I)
40 NEXT I
50 DATA ONE, TWO, THREE, FOUR, FIVE
60 DATA SIX, SEVEN, EIGHT, NINE, TEN
70 INPUT "A NUMBER (1 TO 10) ";N
80 IN N <> INT(N) THEN GOTO 70
90 IF N < 1 OR N > 10 THEN GOTO 70
100 PRINT E$(N)
110 END
```

Design no more than 4 set of test data to FULLY test the above program in responding to the INPUT statement in line 70.

5. Assume that the following program for finding average is correct.

```
10 A = 0
20 C = 0
30 READ X
40 IF X = 0 THEN GOTO 80
50 A = A + X
60 C = C + 1
70 GOTO 30
80 B = A/C
90 PRINT B
95 DATA _____
99 END
```

Which of the following DATA satement will cause error or warning to the above program when it is being executed ?

A. 95 DATA 1,2,3,4,5,0
B. 95 DATA 0
C. 95 DATA -1,-2,-3,-4,0
D. 95 DATA 999,0
E. 95 DATA 1.2,2.3,2,5,9.3,0

Modify one or two statements in the above program to eliminate the error or warning found.

6. Assume that the following program can find the largest integer among the inputted integral values (more than 1) except 999 which is a marker for termination.

```
10 M = 0
20 INPUT X
30 IF X = 999 THEN GOTO 60
40 IF X > M THEN M = X
50 GOTO 20
60 PRINT "THE LARGEST VALUE IS ";M
70 END
```

Which of the following sequence of inputted values to line number 20 will cause incorrect output ?

A. 3,4,999
B. -3,4,5,999
C. 5,4,3,999
D. -4,-3,999
E. 4,1000,999

Modify one or two statements in the above program to eliminate the error found.

7. Assume that the following is a correct sorting program.

```
10  DIM A(100)
20  READ N
30  FOR I = 1 TO N
40     READ A(I)
50  NEXT I
60  FOR I = 1 TO N - 1
70     FOR J = 1 TO N - 1
80        IF A(J) <= A(J + 1) THEN GOTO 120
90        T = A(J)
100       A(J) = A(J + 1)
110       A(J + 1) = T
120    NEXT J
130 NEXT I
140 FOR I = 1 TO N
150    PRINT A(I)
160 NEXT I
170 END
```

Which of the following DATA statement will cause incorrect output ?

A. 55 DATA 3, -1, -2, -3
B. 55 DATA 1, 1
C. 55 DATA 2, 1, 2
D. 55 DATA 3, 3, 2, 1
E. 55 DATA 3, 0.1, 0.2, 0.3

<u>Add one or two statements in the above program</u> to eliminate the above error found.

8. A student writes the following program to compare pairs of numbers and print out a message for each pair stating which number was the bigger.

```
10 READ A;B
20 IF A = -1 THEN GOTO 120
30 IF A > B THEN GOTO 60
40 PRINT B; "IS BIGGER"
50 GOTO 10
60 PRINT A; "IS BIGGER"
70 GOTO 10
80 DATA 5,7
90 DATA 7,5
100 DATA 3,3
110 DATA -1
120 END
```

(a) When the program was executed, a syntax error was reported. Which line of the program contains this error and what should the line be ?

(b) When the program including the correction you have made in (a) was run, an execution error was reported by the computer. Dry run the program to find out which line has caused the error. Write down any changes which are needed to correct this error.

(c) When the program including both your suggested corrections was run again, a logical error occured, write down any changes which are needed to correct this error.

*** END OF THIS TEST ***

(4) TEST ON PROBLEM SOLVING IN PROGRAMMING

INSTRUCTIONS

1. Write down your name, class, and class number on the answer sheet provided.

2. There are TWO questions. Answer ALL questions.
   You have 1 HOUR AND 10 MINUTES.
   You are advised to spend 30 minutes on the first question and 40 minutes on the second one.

3. You are advised to READ the whole question and PLAN before writing each program.

A Partial Character list for ASCII

Character	ASCII	Character	ASCII
0	48	F	70
1	49	G	71
2	50	H	72
3	51	I	73
4	52	J	74
5	53	K	75
6	54	L	76
7	55	M	77
8	56	N	78
9	57	O	79
:	58	P	80
;	59	Q	81
<	60	R	82
=	61	S	83
>	62	T	84
?	63	U	85
@	64	V	86
A	65	W	87
B	66	X	88
C	67	Y	89
D	68	Z	90
E	69		

List of operators and Reserved Words

+, -, *, /, ^

>, <, =, >=, <=, <>

AND, OR, NOT

SQR, INT, RND, ABS, SIN, COS, TAN, ATN, LOG, EXP

+(string concatenation), LEN, ASC, VAL, STR$, LEFT$, MID$, RIGHT$, STRING$

PRINT, PRINT TAB, INPUT, DATA, READ, RESTORE, PRINT USING

DIM, STOP, END, GOTO, GOSUB, RETURN, FOR ... TO ... STEP, NEXT, REM, IF ... THEN, IF ... THEN ... ELSE, ON ... GOSUB, ON ... GOTO, ON ERROR

CLS, HOME

1. A student ENCODES a message with 19 characters including spaces, "COMPUTERS ARE FUN !", into 19 numbers. They are recorded in the DATA statements 60 & 70 below.

```
50 DATA 19
60 DATA 334,384,105,222,61,2,175,228,114,235
70 DATA 241,213,206,3,321,152,214,137,224
```

You are asked to write a program to DECODE the message. A subroutine is given to you and it should be used :

```
1200 REM ----- SUBROUTINE 1 -----
1210 FOR I = 1 TO N - 1
1220    FOR J = 1 TO N - I
1230       IF A(J) <= A(J + 1) THEN GOTO 1270
1240       T = A(J)
1250       A(J) = A(J + 1)
1260       A(J + 1) = T
1270    NEXT J
1280 NEXT I
1290 RETURN
```

Your program should do ALL of the following to DECODE the message (No need to COPY the DATA statements and the codes in subroutine 1 into your answer) :

(a) (using line nos. 100 to 190) READ the number of code into a numeric varaible N and the codes into a 40-element numeric array, A(1), A(2), A(3),...;

(b) (using line nos. 200 to 290) SORT array A in ascending order;

(c) (using line nos. 300 to 390) REVERSE the digits of all 3-digit numbers in array A; (e.g. 147 --> 741)

(d) (using line nos. 400 to 490) SORT array A in ascending order;

(e) (using line nos. 500 to 590) ADD 150 to those numbers greater than or equal to 450, and add 100 to those numbers less than 450 in array A;

(f) (using line nos. 600 to 690) REVERSE the digits of all 3-digit numbers in array A;

(g) (using line nos. 700 to 790) SORT array A in ascending order;

(h) (using line nos. 800 to 890) SUBTRACT 113 from each numbers in array A;

(i) (using line nos. 900 to 990) SUBTRACT 28 repeatedly from each number in array A until the number is less than 28;

(j) (using line nos. 1000 to 1090) ADD 65 to those number smaller than or equal to 25 and add 6 to those numbers greater than 25 in array A;

(k) (using line nos. 1100 to 1190) use the values in array A as the decimal form of the ASCII, CONVERT the codes into characters. Use a string varaible B$ to CONCATE all the characters. PRINT B$ in a line and end the program.

(Put any other subroutines, if used, behind line 1300.)

2. A shop uses computers to handle (處理) sales. A sample VDU output is shown below. In this sample, all the information following the question marks is entered by the cashier (收銀員) through the keyboard. All other output items are from the program.

```
STOCK CODE  ? 043
QUANTITY    ? 1
STOCK CODE  ? 129
QUANTITY    ? 2
STOCK CODE  ? 000
TOTAL COST IS 34.9.              CASH ? 100

HONG KONG SUPPLIES LIMITED

STOCK CODE      PRICE           QUANTITY        COST
043             29.9            1               29.9
129             2.5             2               5
TOTAL     34.9
CASH      100
CHANGE    65.1


STOCK CODE  ? 086
NO SUCH STOCK CODE
STOCK CODE  ? 087
QUANTITY    ? 2.5
MUST BE A POSITIVE INTEGER
QUANTITY    ? 1
STOCK CODE  ? 000
TOTAL COST IS 56.4.              CASH ? 50
NOT ENOUGH CASH
TOTAL COST IS 56.4.              CASH ? 56.4

HONG KONG SUPPLIES LIMITED

STOCK CODE      PRICE           QUANTITY        COST
087             56.4            1               56.4
TOTAL     56.4
CASH      56.4
CHANGE    0


STOCK CODE ? BYE
```

In this shop, there are NO MORE THAN 80 stocks to be sold. Every morning, a table of stock code (貨品編號) and unit price (單價) for all items to be sold on that day are prepared. They are recorded as shown in the DATA statements below with "XXX" as the reading terminator (讀完標記):

```
1010 DATA "128",12.6
1020 DATA "043",29.9
1030 DATA "001",3.7
  :          :
             :
1790 DATA "129",2.5
1800 DATA "087",56.4
1810 DATA "XXX",0
```

The stock codes are 3-digit numeric codes and they are unique (不重複)in the table. Their corresponding prices are given to 1 decimal place.

Each morning, the DATA table are read once. When a customer buy some stocks, the cashier will input the stock codes and quantities (數量) for each stock. The program will Search the price for each stock SEQUENTIALLY, calculate the total cost and ask for the cash of the transaction (交易).

Assume that "000" is a terminating marker (完結標記) for a transaction (交易) and "BYE" is an ending marker of the program (程式完結標記), write a BASIC program to handle sales in this shop and produce the output as shown. Validate the data inputted as shown in the sample output, other validations are NOT necessary.

VARIABLES

Use the following defined variables. You need to use other variables as well which should be defined by yourself.

SC$( )	Stock Codes from DATA table
PC( )	Prices from DATA table
N	Number of stock codes in the DATA table
M	Number of stock codes entered in a certain transaction
T	Total cost
CA	Cash
CG	Change

No default value of the variables in the machine should be assumed.

END OF PAPER

QUESTIONAIRE ON SOLVING PROGRAMMING PROBLEMS

SERIAL NO. :____

Class : _____ No.: (____) Name : _____

Problem 1 : *Decoding*

1. Do you read the whole question completely BEFORE CODING ?

   (Y/N) _____ If yes, how many times ? Roughly ____ times.

2. Have you PLANNED to use other subroutine beside the given one BEFORE you CODE THE PROGRAM ?

   (Y/N) _____

Problem 2 : *Point-of-sale*

3. HOW MANY TIMES have you READ the QUESTION before you start writing your program ?

   Roughly _____ times.

4. Have you PLANNED your program, like the using of additional arrays beside the given ones, constructing the control flow of the program, etc, before coding ?

   (Y/N) _____ If no, no need to answer Q.5 below.

5. If your answer in Q.4 above is yes, what tools have been used to help your planning ?

   (i) Flowcharting (Y/N) ____
   (ii) Stepwise refinement (Y/N) ____
   (iii) Pseudo-code (Y/N) ____
   (iv) Others (please specify) _____
   _____

THANK YOU VERY MUCH FOU YOUR PARTICIPATION IN THIS RESEARCH !

APPENDIX E

(1) *ANSWER SHEET ON LANGUAGE PROFICIENCY TEST*

Serial No. : _____

Class : ___ No. : (___) Name : _____

Personal Particulars

Date of birth : ___/___/___ Sex : _____ Home Computer : _____
DD MM YY M/F Y/N

No. of student per teacher : _____
(1-40)

ANSWERS

1.____ 11.____

2.____ 12.____

3.____ 13.____

4.____ 14.____

5.____ 15.____

6.____ 16.____

7.____ 17.____

8.____ 18.____

9.____

10.____

(2) ANSWER SHEET ON TEMPLATE MEASUREMENT TEST

Serial No. : _____

Class : ___ No. : (___) Name : _____

1. ____
2. ____
3. ____
4. ____
5. ____
6. ____
7. ____
8. ____
9. ____
10. ____
11. ____
12. ____
13. ____
14. ____
15. ____
16. ____
17. ____
18. ____

(3) ANSWER SHEET ON TEST ON PROCEDURAL SKILLS

Serial No. :______

Class : ___ No. : (___) Name : ______________________________

1. (i) ____________________
   (ii) ____________________
   (iii) ____________________
   (iv) ____________________

2. 130 DATA ____

3. (i) |_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
   (ii) |_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
   (iii) |_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
   (iv) |_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|

4. (i) ________________________________
   (ii) ________________________________
   (iii) ________________________________
   (iv) ________________________________

5. Answer on M.C. __________

6. Answer on M.C. __________

7. Answer on M.C. __________

8. Line number Correction

(a) ____________ ______________________________

(b) ____________ ______________________________

(c) ____________ ______________________________

(4) ANSWER SHEET ON PROBLEM SOLVING IN PROGRAMMING

BASIC Used : Microsoft/Apple/BBC Serial No. : ____

Class : ___ No. : (___) Name : _____

1.

LINE NO.	PROGRAM
0 1 2 3 4	5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39

2.

LINE NO.	PROGRAM
0 1 2 3 4	5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39

(5) ANSWER SHEET FOR ADVANCED PROGRESSIVE MATRICES

Serial No. : _____

Class : ___ No. : (___) Name : _____

Answers to sample questions

1. 8 2. 1 3. 3 4. 6

ANSWERS

(1)____ (16) ____

(2)____ (17) ____

(3)____ (18) ____

(4)____ (19) ____

(5)____ (20) ____

(6)____ (21) ____

(7)____ (22) ____

(8)____ (23) ____

(9)____ (24) ____

(10)____ (25) ____

(11)____ (26) ____

(12)____ (27) ____

(13)____ (28) ____

(14)____

(15)____

APPENDIX F

Key to the Language Proficiency Test

Data Structure	Control Structure	Execution Structure
1. C	7. C	13. E
2 C	8. B	14. C
3. A	9. D	15. E
4. B	10. A	16. D
5. D	11. B	17. B
6. E	12. E	18. A

Number of A = 3
Number of B = 4
Number of C = 4
Number of D = 3
Number of E = 4

Key to the Template Measurement Test

Low level Template	Intermediate level Template	High level Template
1. E	7. D	13. C
2. B	8. A	14. D
3. D	9. E	15. C
4. C	10. D	16. A
5. C	11. B	17. E
6. B	12. A	18. A

Number of A = 4
Number of B = 3
Number of C = 4
Number of D = 4
Number of E = 3

(3) ANSWER SHEET ON TEST ON PROCEDURAL SKILLS

Class : _____ No. : (_____) Name : _____

1. (i) 1
   (ii) -1
   (iii) 0
   (iv) 999

2. 130 DATA 1,40,50,70,80,90,99,0

3. (i) "" or " "
   (ii) "TEST"
   (iii) "THIS IS A TEST"
   (iv) "THIS IS A TEST"

4. (i) -1
   (ii) 11
   (iii) 2.4
   (iv) 3

5. Answer on M.C. (B)
   40 IF X = 0 THEN GOTO 75
   75 IF C = 0 THEN PRINT "NO AVERAGE":GOTO 99
   or
   80 IF C > 0 THEN B = A/C
   85 IF C = 0 THEN PRINT "NO AVERAGE":GOTO 99

6. Answer on M.C. (D)
   10 INPUT M or
   10 M = -65535 or other large negative numbers

7. Answer on M.C. (B)
   55 IF N = 1 THEN GOTO 140

8.

	Line number	Correction
(a)	10	10 READ A,B
(b)	10	10 READ A 25 READ B or
	110	110 DATA -1,0
(c)	Add 35 or 25	35 IF A = B THEN PRINT "THEY ARE EQUAL":GOTO 10

APPENDIX G
MARKING SCHEME ON PROCEDURAL SKILLS

1.	A positive real number	1 mark
	A negative real number	1 mark
	0	1 mark
	999	1 mark
2.	Any 3 integers satisfying the 3 cases	1 mark
	1, 50, 80 and 99 testing all boundaries	1 mark
	3 integers fall inside the 3 intervals	1 mark
	A zero as last element	1 mark
3.	A null string or string with spaces only	1 mark
	A string with a single word	1 mark
	A string with more than a word	1 mark
	A string with a leading, or trailing space or more than one space between words which may coincide with the above cases	1 mark
4.	An integer from 1 to 10	1 mark
	A decimal number	1 mark
	An integer smaller than 1	1 mark
	An integer larger than 10	1 mark
5.	M.C. incorrect	0 mark awarded
	M.C. correct with no or completly irrelevant changes	1 mark awarded
	M.C. correct and try to fix error but do not use C as indicater for change or use more than 2 statements to fix error	2 marks awarded
	M.C. correct and fix error correctly but print B as 0	3 marks awarded
	M.C. correct and fix error correctly but do not print B or PRINT "NO AVERAGE"	4 marks awarded
6.	M.C. incorrect	0 mark awarded
	M.C. correct with no or completely irrelevant changes	1 mark awarded
	M.C. correct and try to fix error discovered but not completely correct	2 marks awarded
	M.C. correct and fix error correctly but use very complicated method using more than 2 statements	3 marks awarded
	M.C. correct and fix error correctly	4 marks awarded
7.	M.C. incorrect	0 mark awarded
	M.C. correct with no correct changes	1 mark awarded
	M.C. correct and check N <= 1 to avoid invalid entry of FOR/NEXT loop but cause other errors like READ improperly	2 marks awarded
	M.C. correct and check N <= 1 to avoid invalid entry of FOR/NEXT loop and READ data properly but do not PRINT result properly like A(1) or "NO sorting needed"	3 marks awarded
	M.C. correct and check N <= 1 to avoid invalid entry of FOR/NEXT loop and READ data and PRINT result of sorting properly	4 marks awarded
8.	(a) Only accept change to 10 READ A, B and not others for removal of SYNTAX ERROR	1 mark
	(b) Removal of the execution error	1 mark
	(c) Removal of the logical error for A = B	2 marks

MARKING SCHEME ON PLANNING ABILITY

1. Decoding

a) Call sorting subroutine	4 marks
b) Construct reverse subroutine	4 marks
Call reverse subroutine	2 marks
Total :	10 marks

2. Point-of-sale

a) Flow of control	
Checking S$="BYE" and jump to end	1 mark
Checking S$="000" and jump to input cash	1 mark
Jump to input next S$ after input stock & quantity	1 mark
Jump to input next transaction when one is finished	1 mark
b) Initialization for array dimension	1 mark
c) An array to store quantity	1 mark
An array to store indexing for stock & price (or an array to store stock & price)	1 mark
d) Counting M for number of items brought	1 mark
e) Decompose Stock Code input, Quantity Input & Total Cost calculation into 3 distinct modules	2 marks
Total:	10 marks

3. Traffic survey

a) Decompose the problem clearly into IPO modules	2 marks
b) Initialize variables	
Counters (SP, NV, VF)	1 mark
Largest value (LP)	1 mark
c) Control flow	
Check SG = o for ending of input	1 mark
Check SG = 1 for counting VF & SP and wait for next input	1 mark
Check SG = 2 for counting NV and find largest LP and wait for next input	1 mark
d) Reset VF for finding largest LP	1 mark
e) Find largest LP for the last set of vehicle-free data	1 mark
f) Find average AV after all signals have been inputted	1 mark
Total :	10 marks

MARKING SCHEME ON PROBLEM SOLVING ON PROGRAMMING

```
10 REM DECODING SECRET CODE (APPLE VERSION)
20 REM
30 REM ----- SECRET CODES -----
50 DATA 10
60 DATA 334,384,105,222,61,2,175,228,114,235
70 DATA 241,213,206,3,321,152,214,137,224
80 REM
100 REM ----- READ SECRET CODE -----
110 DIM A(40)
120 READ N
130 FOR I = 1 TO N
140     READ A(I)
150 NEXT I
210 GOSUB 1210 : REM SORTING
310 GOSUB 1310 : REM REVERSE
410 GOSUB 1210 : REM SORTING
500 REM ----- ADD 150 OR 100 -----
510 FOR I = 1 TO N
520   IF A(I) >= 450 THEN A(I) = A(I) + 150
530   IF A(I) <  450 THEN A(I) = A(I) + 100
540 NEXT I
610 GOSUB 1310 : REM REVERSE
710 GOSUB 1210 : REM SORTING
800 REM ----- SUBTRACT 113 -----
810 FOR I = 1 TO N
820     A(I) = A(I) - 113
830 NEXT I
900 REM ----- SUBTRACT 28 -----
910 FOR I = 1 TO N
920     IF A(I) < 28 THEN GOTO 950
930     A(I) = A(I) - 28
940     GOTO 920
950 NEXT I
1000 REM ----- ADD 65 OR 6 -----
1010 FOR I = 1 TO N
1020   IF A(I) <= 25 THEN A(I) = A(I) + 65:GOTO 1040
1030   IF A(I) > 25  THEN A(I) = A(I) + 6
1040 NEXT I
1100 REM ----- PRINT CHR$ -----
1110 B$ = ""
1120 FOR I = 1 TO N
1130     B$ = B$ + CHR$(A(I))
1140 NEXT I
1150 PRINT B$
1160 END
1200 REM ----- SORTING SUBROUTINE -----
1210 FOR I = 1 TO N - 1
1220 FOR J = 1 TO N - I
1230     IF A(J) <= A(J + 1) THEN GOTO 1270
1240     T = A(J)
1250     A(J) = A(J + 1)
1260     A(J+1) = T
1270 NEXT J
1280 NEXT I
1290 RETURN
1300 REM ----- REVERSE SUBROUTINE -----
1310 FOR I = 1 TO N
1320     A$ = STR$(A(I))
1325     IF LEN(A$) <> 3 THEN 1350
1330     A$ = RIGHT$(A$,1) + MID$(A$,2,1) + LEFT$(A$,1)
1340     A(I) = VAL(A$)
1350 NEXT I
1360 RETURN
```

1 mark
1 mark
1 mark (with RETURN)
1 mark
1 mark (with or without RETURN)

COMPUTERS ARE FUN !

Total : 33 marks

```
10 REM DECODING SECRET CODE (GENERAL VERSION)
20 REM
30 REM ----- SECRET CODES -----
50 DATA 19
60 DATA 334,384,105,222,61,2,175,228,114,235
70 DATA 241,213,206,3,321,152,214,137,224
80 REM
100 REM ----- READ SECRET CODE -----
110 DIM A(40)
120 READ N
130 FOR I = 1 TO N
140    READ A(I)
150 NEXT I
210 GOSUB 1210 : REM SORTING
310 GOSUB 1310 : REM REVERSE
410 GOSUB 1210 : REM SORTING
500 REM ----- ADD 150 OR 100 -----
510 FOR I = 1 TO N
520    IF A(I) >= 450 THEN A(I) = A(I) + 150
530    IF A(I) <  450 THEN A(I) = A(I) + 100
540 NEXT I
610 GOSUB 1310 : REM REVERSE
710 GOSUB 1210 : REM SORTING
800 REM ----- SUBTRACT 113 -----
810 FOR I = 1 TO N
820    A(I) = A(I) - 113
830 NEXT I
900 REM ----- SUBTRACT 28 -----
910 FOR I = 1 TO N
920    IF A(I) < 28 THEN GOTO 950
930    A(I) = A(I) - 28
940    GOTO 920
950 NEXT I
1000 REM ----- ADD 65 OR 6 -----
1010 FOR I = 1 TO N
1020    IF A(I) <= 25 THEN A(I) = A(I) + 65:GOTO 1040
1030    IF A(I) > 25  THEN A(I) = A(I) + 6
1040 NEXT I
1100 REM ----- PRINT CHR$ -----
1110 B$ = ""
1120 FOR I = 1 TO N
1130    B$ = B$ + CHR$(A(I))
1140 NEXT I
1150 PRINT B$
1160 END
1200 REM ----- SORTING SUBROUTINE -----
1210 FOR I = 1 TO N - 1
1220 FOR J = 1 TO N - I
1230    IF A(J) <= A(J + 1) THEN GOTO 1270
1240    T = A(J)
1250    A(J) = A(J + 1)
1260    A(J+1) = T
1270 NEXT J
1280 NEXT I
1290 RETURN
1300 REM ----- REVERSE SUBROUTINE -----
1310 FOR I = 1 TO N
1320    IF A(I) < 100 THEN 1350
1325    H = INT(A(I)/100)
1330    R = A(I) - H*100
1340    T = INT(R / 10)
1342    E = A(I) - (H * 100 + T * 10)
1344    A(I) = E * 100 + T * 10 + H
1350 NEXT I
1360 RETURN
```

COMPUTERS ARE FUN !

```
   10 REM !!!!!!!!!!!!!!          POINT-OF-SALE          !!!!!!!!!!!!!!
   30 REM ----- INITIALIZATION ----------------------------------------
2  40 DIM SC$(80),PC(80)
   42 DIM S$(80),P(80),Q(80),C(80)
  100 REM ----- READ STOCK CODE & PRICE -------------------------------
  110 I = 1
  120 READ SC$(I),PC(I)
  130 IF SC$(I) = "XXX" THEN GOTO 160
4 140 I = I + 1
  150 GOTO 120
  160 N = I - 1
  200 REM ----- INPUT STOCK CODE & SEARCH FOR PRICE -------------------
  210 M = 0
  220 INPUT "STOCK CODE  ";S$
  223 IF S$ = "BYE" THEN GOTO 2000
  225 IF S$ = "000" THEN GOTO 410
  250 FOR I = 1 TO N
10 2  260    IF SC$(I) = S$ THEN GOTO 290
  270 NEXT I
  280 PRINT "NO SUCH STOCK CODE" : GOTO 220
  290 M = M + 1
  295 S$(M) = S$
  296 P(M) = PC(I)
  300 REM ----- INPUT QUANTITY ----------------------------------------
  310 INPUT "QUANTITY    ";Q
5 320 IF Q<=0 OR Q<>INT(Q) THEN PRINT "MUST BE POSITIVE INTEGER":GOTO 310
  330 Q(M) = Q
  340 GOTO 220
  400 REM ----- CALCULATE COST AND TOTAL COST ------------------------
  410 T = 0
  415 IF M = 0 THEN GOTO 695
4 420 FOR I = 1 TO M
  430    C(I) = P(I) * Q(I)
  440    T = T + C(I)
  450 NEXT I
  500 REM ----- INPUT CASH --------------------------------------------
  510 PRINT "TOTAL COST IS ";T,"CASH ";
3 515 INPUT CA
  520 IF CA < T THEN PRINT "NOT ENOUGH CASH" : GOTO 510
  530 CG = CA - T
  600 REM ----- PRINT OUT ---------------------------------------------
  610 PRINT : PRINT
  615 PRINT "HONG KONG SUPPLIES LITMITED"
  620 PRINT
  630 PRINT "STOCK CODE", "PRICE", "QUANTITY","COST"
5 640 FOR I = 1 TO M
  650    PRINT S$(I),P(I),Q(I),C(I)
  660 NEXT I
  670 PRINT "TOTAL  ";T
  680 PRINT "CASH   ";CA
  690 PRINT "CHANGE ";CG
  692 PRINT : PRINT : PRINT
  695 GOTO 210
 1000 REM ----- STOCK CODE & PRICE TABLE ------------------------------
 1010 DATA "128",12.6
 1020 DATA "043",29.9
 1030 DATA "001",3.7
 1040 DATA "129",2.5
 1050 DATA "087",56.4
 1060 DATA "XXX",0
 2000 END
```

total : 53 marks

```
   10 REM !!!!!!!!!!!!!!          POINT-OF-SALE          !!!!!!!!!!!!!!
   30 REM ----- INITIALIZATION ----------------------------------------
2  40 DIM SC$(80),PC(80)
   42 DIM I(80),Q(80)
  100 REM ----- READ STOCK CODE & PRICE -------------------------------
  110 I = 1
  120 READ SC$(I),PC(I)
  130 IF SC$(I) = "XXX" THEN  GOTO 160
  140 I = I + 1
  150 GOTO 120
  160 N = I - 1
  200 REM ----- INPUT STOCK CODE & SEARCH FOR PRICE -------------------
  210 M = 1
  215 T = 0
  220 INPUT "STOCK CODE  ";S$
  223 IF S$ = "BYE" THEN GOTO 2000
  225 IF S$ = "000" THEN M = M - 1 : GOTO 505
3 250 FOR I = 1 TO N
  260    IF SC$(I) = S$ THEN I(M) = I : GOTO 310
  270 NEXT I
  280 PRINT "NO SUCH STOCK CODE" : GOTO 220
  300 REM ----- INPUT QUANTITY ----------------------------------------
  310 INPUT "QUANTITY    ";Q
+ 2 320 IF Q<=0 OR Q<>INT(Q) THEN PRINT "MUST BE POSITIVE INTEGER":GOTO 310
  330 Q(M) = Q
  400 REM ----- CALCULATE COST AND TOTAL COST -------------------------
  2 410 T = T + PC(I(M))*Q(M)
+ 1 420 M = M + 1
  430 GOTO 220
  500 REM ----- INPUT CASH --------------------------------------------
  505 IF M = 0 THEN GOTO 685
  510 PRINT "TOTAL COST IS ";T,"CASH ";
+ 515 INPUT CA
  520 IF CA < T THEN PRINT "NOT ENOUGH CASH" : GOTO 510
  530 CG = CA - T
  600 REM ----- PRINT OUT ---------------------------------------------
  610 PRINT : PRINT
  615 PRINT "HONG KONG SUPPLIES LITMITED"
  620 PRINT
  630 PRINT "STOCK CODE", "PRICE", "QUANTITY","COST"
  640 FOR I = 1 TO M
2 650    PRINT SC$(I(I)),PC(I(I)),Q(I),PC(I(I))*Q(I)
  660 NEXT I
  670 PRINT "TOTAL  ";T
  680 PRINT "CASH   ";CA
  690 PRINT "CHANGE ";CG
  692 PRINT : PRINT : PRINT
  695 GOTO 210
 1000 REM ----- STOCK CODE & PRICE TABLE -----------------------------
 1010 DATA "128",12.6
 1020 DATA "043",29.9
 1030 DATA "001",3.7
 1040 DATA "129",2.5
 1050 DATA "087",56.4
 1060 DATA "XXX",0
 2000 END
```

Total : 33 marks

APPENDIX H

THE INTERVIEW SCHEDULE

Objective

The interviews conducted in this research are to supplement the quantitative method applied to study the behavior of problem solving in programming in secondary school students. The interviews serve to go deeper into the *motivations* of respondents and their *reasons* for responding as they do in the programming tests conducted in this research.

Type of interview

There are two broad types of interview, namely, structured and unstructured interview. In the structured interview, the questions, their sequence, and their wording are fixed. Unstructured interviews are more flexible and open. Although the research purposes govern the questions asked, their content, their sequence, and their wording are entirely in the hands of the interviewer. In accordance with the nature of the interview in this research, a compromise type of interview is concluded in which the questions are carefully planned but the interviewer is permitted leeway to use alternate questions that he judges fit particular respondents and particular questions. The schedule items are open-end questions. The interview starts with a broad general question and narrows down progressively to the important specific points (Kerlinger, 1973).

The Subjects

The interviewees are chosen by their teachers under the requirement of the researcher that they are slightly below average among their classmates. The subjects are so chosen to maximize the information to be obtained because low achievers may be unable to express their problems in programming properly while high achievers may think in the way as an expert do and their information may not be as fruitful as average students may provide.

The interview schedule

The questions designed in the interview serve to explore how secondary students tackle programming problems and what knowledge and skills are involved in the process. The interview may assist to validate the programming model proposed in the theoretical framework of the paper.

Three central themes of the programming process will be discussed surrounding each programming problem, namely, understanding the question, planning and coding, and, testing and reformulating. The following are the detailed questions asked for each of the three perogramming problems. The general questions will be asked first and the detailed questions will be followed to guide the students to respond to the main question.

1. Decoding (30 minutes)

I) Do you understand the question ? (5 minutes)

1) Can you tell what the question is asking ?
2) Which part is most difficult for you to understand ?

II) You plan before coding or planning in action ? (15 minutes)

1) What is the function of the given routine ?
2) Do you use the given routine ? If not, why ?
3) Do you recognize that their are common functions among the processing requirements ? When do you notice it ? Any action taken ?
4) Can you give reason(s) why subroutine are used in general ?
5) Is GOSUB/RETURN familiar to you ? Do you use it commonly ?
6) How do you concieve the codes for reversing the order of the numeric code ? Have you written these codes before ?
7) Which part is most difficult to you in coding ? Why ?
You code it for the first time ?
8) Which part is easy to you in coding ? Why ?
Do you write these codes frequently ?

III) Have you test and reformulate your program ? (10 minutes)

1) Do you put test data to test your program or part of it ?
2) Have you checked whether your reverse module can serve for non 3-digit numeric codes ?
3) Have any part been reformulated in this program ?
4) Do you spent time on designing test data to test your programs in general ?
Can you tell some criteria in setting test data ?
5) Can you fix the errors by yourself, in general ?
Can you tell the most common method you use in reformulating programs when error are found ?

2. Point-of-sale (30 minutes)

I) Do you understand the question ? (5 minutes)

1) Can you tell what the question is asking ?
2) Which part is most difficult for you to understand ?

II) You plan before coding or planning in action ? (20 minutes)

1) How the control flow of the program is constructed ?
e.g. Checking S$="BYE" and jump to end
Checking S$="000" and jump to input cash
Jump to input next S$ after input stock & quantity
Jump to input next transaction when one is finished
What tool is used, if any ? Can you show it ?
Do you use this method frequently ? Is it a good method ?
Any suggested better methods ?
2) Are additional arrays introduced ? Why not ?
What are they, if used ?
e.g. An array to store quantity
An array to store indexing for stock & price
(or an array to store stock & price)
Do you initialize them ? Is it needed ?
Do you forget to initialize array frequently ?
3) What is the relationship between Stock Code input, Quantity Input & Total Cost calculation ?
You prefer to write the codes independently or mix together ?
4) Can you write sequential search immediately as required in this question ? What is its function in this question ?
5) Can you write statements for counting immediately ?
Can you write statements for keeping a running total ?
6) Can you write statements for checking whether the inputs are positive integers ?
7) Can you read a set of data with a reading terminator ?
Can you print a table of outputs ?
8) Are the knowledge in (4) to (7) helpful to you ?
Do you apply them in your planning ?
Do you apply them in your coding ?

III) Do you test and reformulate your program ? (5 minutes)

1) Do you put test data to test your program or part of it ?
2) Have you ever think about M = 0 ?
This will occur when "000" is entered accidentally with no other inputs ?
Do special treatments needed ?
3) Have any part been reformulated in this program ?
4) Do you spent time on designing test data to test your programs in general ?
Can you tell some criteria in setting test data ?
5) Can you fix the errors by yourself, in general ?
Can you tell the most common method you use in reformulating programs when error are found ?

APPENDIX I

PROGRAMMING PROTOCOLS

The following transcripts were translated from Chinese.

Subject A - Male Form 5 student

S: (Read problem Point-Of-Sale)
Pause

R: Describe the question that you have read.

S: A shop has a computer. Question requires me to input stock code and price. Computer do the calculation and print out stock code, price and changes.

R: How to tackle the program at first glance ?

S: Read question, don't understand, read sample outputs, getting to know input data to the computer for printing a bill for a sale. Read the question again, do as required, no flowcharting is required.
Pause

S: Compare sample outputs and wordings in question to see the requirement. Much time is spent in understanding the question.

S: Read data, one by one. Use an array to store stock code SC$( )and price PC$( ). When "XXX" is encountered in reading stock code, it means end of reading. Another part can be started. This part repeats everyday.

S: If someone buy things, the program will ask for stock code. Enter stock code. Program check whether length of stock code is 3. If no, it should be rejected. After reading sample output, I know the checking is not required. Its length must be three. Then, use M$ to compare with SC$( ), if found then price can be found, if not found then inform no such stock. If found, input quantity. It must be integer. It must be tested. If not integral, reenter.

S: After quantity is inputted, another stock code can be inputted. If "000" is encountered, it means end of input.

Then add all prices, print total price, ask for cash, and print changes. List the bill.

R: Any more ?

S: Check whether there are bugs in the program.

R: How to check ?

S: Use the computer to check.

S: If I have time, I shall print the output more neatly.
Pause

S: Add one more statement checking whether stock code is "Bye". If yes, the program end.

R: Can you write out the codes now ?

S: Write from the beginning ?

R: Yes

Subject started to write the program and thinking aloud as required.

S: Read the data first. Write a REMark statement to indicate read which is convenient for later reference. Reserve 80 storages for SC$( ), PC( ) and M$( ) using DIMension statement.

S: FOR/NEXT loop cannot be used for reading codes because number of codes is not known at the beginning. Use I as counter. Read SC$( ) and then PC( ). If SC$( ) is "XXX", then finish reading and GOTO line 200, else add 1 to counter I and GOTO line 130. All data can then be read.
Pause

S: From line 200 onwards, use counter I to count number of stock codes inputted from keyboard.
Pause

S: M$ stores stock code just inputted from keyboard. Check whether M$ is one in SC$( ). Uhmmm, counter I cannot be reused becasue it has been used to store number of stock code in data table.
Pause for thinking.

S: I cannot be used as counter in reading as well. It should be N as given in the variable list (Change I as N from line 120 to 150). Use M as counter for counting stock codes inputted from keyboard and store them in M$(M)and check

them sequentially.
Pause

S: Then use FOR/NEXT loop to check whether M$(M) is in SC$( ). (Write codes on paper)
Pause

R: Why you write GOTO 300 immediately when you are writing line 230 ?

S: This line (230) means if M$(M) is found, then jump out the searching loop, therefore jump to line 300, else check continuously inside the loop. If not found at the end of loop, then it means M$(M) not in the data table, print "NO SUCH STOCK CODE" and jump to line 210 for reenter. (Subject has just used 3 minutes to finish coding of the sequential search)
Pause

R: What are you doing now ?

S: Now, input quantity. Use M, no, use N(I) to store quantity (Add N(80) to DIM statement immediately). 80 is used because there are no more than 80 stocks. Check whether it is an integer (Write on paper immediately to check that the number inputted is an non-negative interger).

R: What are you doing now ?

S: Check whether the inputted number is an integer or not. If no print "MUST BE A POSITIVE INTEGER" and then input again.

S: M=M+1 : GOTO 210
Pause (thinking)

S: line 240 NEXT should not be end of loop, ..., no, it should be end of loop.
Pause (thinking)

S: Lines should be added in the front to check whether M$(M) is "000" or "BYE".
Pause (then add line 216 and 217 on paper)

S: The second part should be finished. The third part is to print the bill and calculate the total cost.
Pause

R: What are you doing ?

S: Now, calculate the total cost.
Pause (thinking)
S: Use FOR/NEXT loop to calculate.
Pause (try to write codes 400 FOR I=1 TO M:410 T=T+PC(..)
Long pause thinking how to calculate total cost and find mistakes.
S: Wrong !
R: What is wrong ?
S: Serious mistake ! It is wrong because I value has not been stored in checking whether M$(M) is one of the SC$(I). Total cost should be calculated before line 330 (Write 321 T = T + PC(I) X N(M))
S: Input one, calculate one and accumulate total cost in T (Subject adds T = 0 on line 200 immediately as well).
Pause (thinking)
R: What are you thinking ?
S: The storage of quantity in N(I) is wrong, it should be N(M) (change on paper)
Pause
S: Print total cost and then ask for cash.
S: If CA < T Then print "NOT ENOUGH CASH", print three blank lines, no, two, then bill can be printed. Copy those from sample output. Print a blank line and then print title. If time is allowed, beautify the outputs. Then print each line of output using FOR I = 1 to M
Pause (thinking)
R: What are you thinking ?
S: Print one by one...
Pause (thinking)
R: What are you thinking now ?
S: Thinking how to print price,... Array should be used to store the prices in the front.
Pause (thinking how to add the array)
S: Add line 236.
Pause (thinking)
R: What are you thinking ?
S: Add 236 P(M) = PC(I)

(Add P(M) to DIM statement immediately)

S: Print M$(I);" ";P(I);" ";N(I);" ";P(I)XN(I)
Print "TOTAL ";T
Print "CASH ";CA
Print "Change ";T - CA
Print 3 blank lines
GOTO 200 and start for next customer's input again.
That's the end.

R: If more time is allowed, what will you do ?

S: If time allow, I shall copy the program again because it is quite in a mess now.

R: If you have a computer, what will you do ?

S: If there is a computer, I shall RUN it on the machine.

R: What will you do if you have a computer in creating this program ?

S: I shall key the program immediately.

R: Will you draft the program as this one first ?

S: No, everytime when I do a program, I shall key in the lines as I am solving it except in examinations.

R: If the program is in the computer, you will test it, what data would you enter in this program to test it ? What criteria will you use in setting test data ?

S: Just like the sample outputs in this question, enter some data.

R: What data ?

S: All possible cases, like "000", "643" and few more.
Pause

S: Enter some stock codes not in the data table like "003".

R: How about enter "000" as the first stock code in your program ?

S: No, because it is meaningless. Some stock codes like "043", "129", then input "000" as the last one. Enter some cash less than the total cost and so on...

S: A GOTO is missed
Pause (Add GOTO 420 in line 430)

R: Finish ?

S: There may still some mistakes.

Pause (Read program again)

S: No mistake.

R: I want to know more about your coding process. In some coding, like N = 1 : N = N + 1, you seems very familiar with them, aren't you ? What do they represent ?

S: Counter

R: Is N = N + 1 must accompany with N = 1 ?

S: Yes

R: Do you need to think about that in coding ?

S: No, because it is always used in programming.

R: Have you written codes in reading a set of data using a reading terminator like "XXX" in this program ?

S: Yes

R: Do you know how to write the codes before you write this part of coding ?

S: Yes, very familiar but not used to write and speak up simultaneouly. It is confusing in doing the two things at the same time. I can be faster if writing the codes alone.

R: Why you still can write the codes so quickly ?

S: Coding for reading like this have been written before.

R: Have you written codes on searching before ?

S: Yes

R: What is the name of the search that you have write in this program ?

S: I know a search named binary search, but that one is difficult to write and mistakes will be committed easily. The one that I have written is easy. It has a name, but I cannot remember it.

R: Have you used it before in programming ?

S: Yes, several times.

R: Do you have an idea in writing the search before the actual coding ?

S: Yes, there is an idea.

R: What's that ?

S: If found, jump out of loop to state found. If not found, continue the loop until end of loop, then really not found.

R: You also write very fast in checking positive integers in checking quantity inputted. You write immediately when you are at there, why ?

S: These checkings have been written many times in programming assignments.

R: What is the function of this conditional N(I) <> ABS(N(I))?

S: Check negative numbers.

R: This compound conditional
N(I) <> INT(N(I)) OR N(I) <> ABS(N(I)) is used to check negative numbers and integral values. Why do you use AND and change it to OR ?

S: AND is wrong. If +1.1, it cannot be checked.

R: When "AND" will be true in logic ?

S: When both conditions are true.

R: If N(I) is 1.1, will these compound conditional be true ?

S: AND is worng ...
Pause (thinking)

S: 1.1 is not ...

R: Now you are thinking for using AND or OR ?

S: I use AND usually in writing this compound conditional for testing positive integral values.
It should be AND... (Change his mind)

S: I am confused.

R: What do you want to do now ?

S: I want to do it in another way.
Pause

S: Do it in two statements.
Pause (Break the compound conditional into two statements as

```
310 If N(I) <> INT(N(I) THEN ?"MUST BE A POSITIVE
    INTEGER" : GOTO 300
320 IF N(I) <> ABS(N(I)) THEN ?"MUST BE A
    POSITIVE INTEGER" : GOTO 300
```

)

R: When you write T = T + PC(I)*N(M) , you add T = 0 immediately, why ?

S: If T = 0 is not added, it will cumulate values in last time.

R: You put T = T + PC(I) X N(M) output the loop, later put

back inside the loop. Have you thought about this before coding ?

S: At first, I want to do it outside the loop. Later, I find it is incorrect to do it outside the loop becasue there is no price information outside the loop so I put it inside.

R: I have observed that whenever you use a new array, you put it in the DIM statement immediately, Why ?

S: Avoid to forget.

R: You print the table quite easily, are you familiar with the coding ?

S: Just follow the sample output, so it is easy.

R: Before coding, do you know how to code the different parts on reading data, process data and output the information ?

S: Codings on many parts have been learnt before.

R: If there is a person input "000" as the first stock code number, what will happen to your program ?

S: The bill will be printed. All outputs will be zeroes. Ask for cash ? Add one line to prevent this to happen. But I have never thought about this problem.

R: Do you think that this situation should be treated ?

S: Yes, I think so.

Pause (thinking)

S: Do that person know about computer ?

R: If that people really do that...

Pause (thinking)

S: Add 1 line, like IF M = 1, then avoid the printing.

R: What are the criteria in testing a program ?

S: Test all possible cases. Test some normal data and then some irregular data. In this program, type stock code with 2 digits or 4 digits or "ABC". All cases will be checked.

R: Do you usually plan your programs before coding ?

S: No.

```
100 REM ** READ THE DATA
110 DIM SC$(80), PC(80), M$(80), N(80), P(M)
120 N = 1
130 READ SC$(N), PC(N)
140 IF SC$(N) = "XXX" THEN 200
150 N = N + 1
160 GOTO 130
200 M = 1 : T = 0
210 INPUT "STOCK CODE ?";M$(M)
215 IF LEN(M$(M) <> 3 THEN GOTO 210
216 IF M$(M) = "BYE" THEN END
217 IF M$(M) = "000" THEN 400
220 FOR I = 1 TO N
230 IF M$(M) = SC$(I) THEN GOTO 300
236 P(M) = PC(I)
240 NEXT I
250 ? "NO SUCH STOCK CODE" : GOTO 210
300 INPUT "QUANTITY ?";N(M)
310 IF N(I) <> INT(N(I)) THEN ?"MUST BE A POSITIVE INTEGER"
    : GOTO 300
315 IF N(I) <> ABS(N(I)) THEN ?"MUST BE A POSITIVE INTEGER"
    : GOTO 300
320 M = M + 1
321 T = T + PC(I)XN(M)
330 GOTO 210
400 ?"TOTAL COST IS ";T;".";
410 ?SPC(20);
420 INPUT "CASH ?";CA
430 IF CA < T THEN  ?"NOT ENOUGH CASH" : GOTO 420
440 ?:?
450 ?"HONG KONG SUPPLIES LIMITES"
460 ?
470 ?"STOCK CODE   PRICE   QUANTITY   COST "
480 FOR I = 1 TO M
490 ?M$(I);"       ";P(I);"      "N(I);"      ";P(I)XN(I)
```

```
500 ?"TOTAL     ";T
510 ?"CASH      ";CA
520 ?"CHANGE    ";T - CA
530 ?:?:?
540 GOTO 200
```

Subject B - Female F.5 Student

S: (Read the question Point-Of-Sale)
R: You may start to describe the question when you are ready.
S: The question said that there is a shop use computer to process sales. The cashier inputs the stock codes according to the format as in the sample output. The format of the bill for output is also given. The restriction on the number of stock codes and many others are also stated. The question also said that if search is required, sequential search should be used. The question also mentioned when the program would be ended. Variable names are also given.
S: The programming solution can be thought of composing of 3 parts applying the input, processing and output concept.
R: Do you use this concept in other programming assignments ?
S: Yes, sometimes.
S: Usually the input and output parts are more easy. So this two parts will be done first.
S: Input requirements must be known for input. Output must follow the format of the bill. Then process can be done step by step from data inputted to the required output.
R: Can you do the question in a more concrete way by writing down the codes ?
S: I shall mark down the key words in the question first.
S: I shall use a subroutine for searching if it is used more than once in the question. It is used only once in this program. So it is not necessary.
R: You have to code the program.
S: New Variables should be written down first if it is necessary or it will be troublesome in a later stage.
S: I shall think of myself as a cashier. There are someone coming to buy somethings for certain quantities. This information will be entered into the computer. Prices must then be searched from the data table. A searching is necessary. The contol flow of the program will be back when the seaching is finished (The subject assume that the

searching is done by a subroutine). Calculate total. Ask for cash. Calculate change. Ask for cash again if not sufficient money is given. Output can be done easily by following the sample output. The price, stock code, and quantity should be printed out on the bill.

R: Yes, please write the program now and thinking aloud as well.

S: Write the program ?!

R: Do you write the program immediately usually when you start to tackle a programming problem ?

S: Think for a while as stated above. No flowchart will be drawn.

R: Just do the program as you usually do.
Pause

R: What are you thinking ?

S: Thinking about how many memory locations should be reserved in the Dimension statement.
Pause (Write line 10 DIM SC$(80), PC(80))

R: What are you thinking ?

S: There are 80 data representing the stock codes and prices. They must be read into arrays.
Pause (Write lines 20, 30, and 50)

S: Data must be read before sales can be started.
Pause

S: Use IF/THEN statement to check the "XXX" terminator. If it is encountered before 80 stock codes are read, then jump out of loop and continue the other parts.
Pause (Add line 40)

S: Input something for making transactions.
Pause (Write line 60 for input stock code)

R: What are you writing for ?

S: Inputting the stock code in SC$(I).
Pause

R: What are you thinking ?

S: Thinking about the I in SC$(I). Can index I be used later for searching. Will it be mixed with the counter I in the reading above ? (The subject ask herself)

S: It should be correct.

S: Input the quantity. (Writing code for input quantity)
Pause

S: There is some trouble ! I don't know whether it should be doing the sequential search first before inputting the quantity or search after entering all data.
Pause for thinking

S: Something should be add ... Well, no need to add.

S: I think the sequential search should be done first. But I don't know the line number (She add GOTO in line 60 without line number). After searching, then enter the quantity. (Then she write line 80)

S: Add line 80 checking if SC$(I) is "000". If it is, then data input will be stopped.
Pause (Writing 80 IF SC$(I) = "000" THEN GOTO 60)

S: No, no, it should be not equal to "000" then jump back. (She change line 80 to IF SC$(I) <> "000" THEN GOTO 60)

S: After searching, it should come back here.

S: Total cost T should be equal to PC(I) times M(I)
Pause (thinking)

R: What are you thinking ?

S: Confusing. If the subscript is I, the total cost cannot be calculated.

S: After flowing back from the subroutine,it is not separated.

R: Not separated from what ?

S: After searching, it is one stock, two stock, the total cost cannot be calculated. There may be more than one cost.
Pause

S: GOTO in line 60 may not be there, it may be here (She deleted GOTO in line 60 and add GOTO in line 70).

R: Have you done searching ?

S: From here, go to search and then come back. Add a statement to calculate a value by multiply price by quantity and add them together (Add 75 T = PC(I) X M(I) and 76 T = T + ...)

S: T equal to T plus what ?
Pause

S: T in line 75 should be T1 and line 76 should be T = T + T1.

That's it.

S: There is total cost after that (Write line 90 to print the total cost).
Pause

R: What are you doing ?

S: Write line 100 to ask for cash (Write line 100)
Pause (Write codes)

R: What are you thinking ?

S: Thinking what to do in the next step ?

S: No validation for input. It should be added.

S: Add line 72 ...

R: What do you want to do ?

S: Check positive integeral value for quantity inputted.
If not correct, input another value again.
Pause (Writing line 72)

S: There are one more validation.

R: What's that ?

S: No such stock. In the search, a flag F should be used to indicate whether the stock can be found. It should be setted to zero at first. If the stock code is found, it should remain zero. If the stock code is not found, then set the flag to one.
(Subject add line 73 IF F = 1 THEN PRINT "NO SUCH STOCK CODE" : GOTO 60)

S: Line 72 and 73 should better be interchanged. But it will be alright if let them remain unchanged.
Pause

S: 110 CG = CA - T

R: What are you doing ?

S: Calculate the changes.

S: After the processing stage, it should be the output stage.

S: Now it should print the bill. How to do that ?
Pause

S: Just print it.
Pause (Write line 120, 130)

R: Are you writing the format of the bill ?
(Write line 140)

S: One, two, three, four,...
R: What are you counting for ?
S: The output positions.
Pause
R: What are you thinking now ?
S: How to output the price, when to start and when to end ? After printing one line, it should jump to the next line if there is another stock code to be printed. If there is no more, then print the total cost and change. I am thinking how to do these things.
Pause
S: Add an IF statement, if a "000" stock code is inputted, then jump away and do not print, else print.
(Subject write line 150 and 160)
S: All variables in line 160 involved I ...
Pause
R: What is not known ?
S: Variable I has not been given before. I is inside a bracket. Here, it only know that it is I but do not know what it has been stored. Do not know what "I" for output. Therefore, it is troublesome.
Pause (thinking)
R: What are you thinking ?
S: What I am thinking are there any BASIC statements can be used to record the past values.
Pause
R: What are you thinking ?
S: The same problem as before, but still no idea.
Pause
S: Cannot continue... if this problem cannot be solved.
R: If you can solve it, how to continue ?
S: If I can solve it, I shall add a line to print the inputted stock codes in the transaction and print other information as well. Then calculate the total cost and the like. Now, I am entangled in it.
R: If this problem is solved, will your program finish ?
S: No, add one more line to jump back.

Pause (Continue to write line 160)

R: What is the M represent ?

S: It represents the quantity for a particular stock. No, I am wrong from the beginning. That is to say I have to correct from the beginning.

R: Originally what does M represent ?

S: Store the quantity of the stock brought by the customer. But I think it is wrong.

R: What is wrong ?

S: It does not look like correct in line 160, so I think it is wrong. I have used this variable (M) incorrectly.

R: You mean several quantities for several stocks are stored in M.

S: Orginally I think that after M is inputted, a value for the total cost of the stock will be calculated immediately. But now all of them are to be recorded here, then M cannot serve this purpose. Therefore it is wrong in using M.
Pause for thinking

S: A FOR/NEXT loop should be added so that a bracket can be added to the variable, then the order of appearance can be recorded for the quantities for different stocks.

R: Where should the FOR/NEXT loop be added ?

S: Between Line 50 and 60 (Write line 55 and 85).
Add (I) to M and T1 in line 70 and 75 respectively, then they can record the values.

R: What is recorded ?

S: The order of appearance.

R: The order of appearance for what ?

S: The order of appearance for buying the stocks.

S: Add a FOR/NEXT loop to print the stocks in line 145 and 170.

S: Add T1(I) to line 160. Ah ! something is missing between line 100 and 110, a blank print statement is missed. Everything will be printed on the same line (line 145 PRINT is added).

S: Add line 180 to print the total cost (Write line 180).

S: Print the cash and the change (Write line 190 and 200).

S: Ah ! One thing is missed. The program cannot end without it. Add to line 85. No, it exists. Add line 86 ? No, it should be line 84, that is before NEXT in line 85. Line 84 is an IF statement to check "BYE" (Write line 84). If "BYE" is encountered, then jump to END of program.

S: I really cannot remember how to code the sequential search. If it is written, then this program will be quite alright.

R: Can you try to write the sequential search ?

Pause

R: What to search ?

S: Search the stock code.

Pause

S: Mumbling...

R: Can you speak up ?

S: Get a stock code. Check it with all using a FOR/NEXT loop. If not found after all have been checked, then left. If found, left. Therefore, both are left the loop. But, the former should inform that the stock code is not found, the later should continue with the other part of the program. But, I am not able to write the codes because the two variables are colliding.

R: Which two ?

S: The inputted stock code and the stock code read from the data table are the same so there is some problem (The subject use the same name SC$( ) for both of them). Therefore I don't know how to write it. This must be changed.

Pause for thinking

S: Change one variable. The variable for storing the stock codes inputted by the cashier should be changed to a new one.

R: What is its new name ?

S: S$(I), what is inside the bracket is not important. All involved should be changed. Line 60, 80, 160, 150, and 84. That's all.

Pause

R: What are you doing ?

S: Start to write the sequential search.
(Write the codes in line 300 and 310)
If SC$(I)=S$(I) then the stock code is equal to the stock code in the data table, jump back to line 75. Add GOTO 300. I think no need to use a "flag". But it can be used. No need to use the flag. No, no, it should be used. If not found, set the flag to 1. If found, set it to 0.
Pause

S: Change line 300 to 305. Add 300 F=1. Change line 310 THEN 75 to THEN F=0 : GOTO 75
(Write line 320 NEXT I)

S: I think it is alright then in the searching.
Add line 330 GOTO 75, no matter it can be found or not.

S: Have I used the "flag" in the front ?
Oh ! Yes, in line 74. If F = 1 then inform there is no such stock code and request for input again and print the bill.
Pause for thinking

S: Most probably this is the end of the program. Add line 1820 END

R: I want to ask you something.

R: Sometimes you code very fast, for example, to check whether the inputted value is an integral value, why are you so fast in writing it ?

S: This BASIC statement is triggered when it is needed.
But will it be a negative number ? (The subject find she has made a mistake). Add an IF statement to check whether it is smaller than 0.
(Subject add line 73)

R: Do you know what is the function of using an array in programming ?

S: Array can be used to reserve storage to store data.

R: You have said that you want to find some statements in BASIC to store previous values. At that moment, you said you are lacking knowledge to do that. But afterward, you add a bracket to a variable to solve the problem. Do you know...

S: Maybe I am wrong there...

R: What's wrong ?

S: I suspect whether the bracket have the function to do that.

R: What function ?

S: Can it be used to find out the price and stock code later in printing the bill ?

R: Yes, what is the function of using array ?

S: Array is most probably used to reserve area to store data and the user can retrieve it later from a certain location.

R: Can it serve the purpose as you intended now ?

S: Ah...yes, unless the variable is different...

R: Which variable should be changed ?

S: Though the values are stored in array but if we want to find out the exact location in an array, we must know what is the value of "I". But I still do not know how to let the value of "I" to be known in a later stage. If it is not known, then the price and stock code cannot be taken out from the array. In that case, it cannot be printed out. That's it.

R: How to solve this problem ?
Pause

S: No idea.
(This problem is left unsolved)

R: When you write line 76 T=T+T1(I) for calculating total cost, you are very fast in producing the codes. Are you very familiar with codes ?

S: I have used this line of code many many times.

R: What is the function of this line ?

S: It can be used to accumulate the total cost.

R: When you read data, you encountered "XXX" as a terminator. Have you ever encountered such a case in reading data ?

S: No, not before this time. Usually, I use a FOR/NEXT loop to read them and expecting a terminating counter, say 80 as in this problem.

R: How do you know the terminating counter, say 80 in this case ?

S: This 80 is given in this problem because there are no more than 80 stock in this shop.

R: In this problem, will it read all 80 stock codes ?

S: May be not.
Pause

R: If this prblem is finished and all syntax errors are corrected, how are you going to test the program ?

S: Input one stock code not in the data table. Input a negative number and a decimal number. Input a "000" as a stock code. Input few stock codes that is in the data table.

R: Why a few ?

S: To make sure that the program can print a bill having a few lines.

R: What is your criteria in setting test data to test a program in general ?

S: Include data in more aspects. First of all is to fulfil the requirements of the question. Then special data like 0, positive or negative values, etc. Also, like "BYE" in this problem.

R: If you find logical errors in your program, how are you going to remove them ?

S: Rethink what is wrong ?

R: What method will be applied ?

S: Read the question again and note the important key words in the question. See whether there are important information missed.

R: Do you think planning a programming solution is useful ?

S: Maybe useful. Usually, I just separate the program into three parts, that is, input, processing and output. The logic of the program mainly develop following the information provided by the question but not any other designing devices.

R: Is it difficult to you to use flowchart to present the logic of this program ?

S: To me, it is difficult because I am used to draw flowchart after the program is coded.

R: Do you think that you have planned before you write the above program ?

S: Yes, a little bit. Separate into parts.
R: Is the planning just construct in the brain ?
S: Yes.
R: Is the plan modified as the code is writing ?
S: In general, no. But when error is detected, it is modified.
R: Do you think that your knowledge in BASIC is sufficient for you to solve problem of this type ?
S: I believe it is not sufficient, still need to refer to text for more detail knowledge in BASIC.

Programming Solution by subject B

```
10 DIM SC$(80), PC(80)
20 FOR I = 1 TO 80
30 READ SC$(I), PC(I)
40 IF SC$(I) = "XXX" THEN GOTO 60
50 NEXT I
55 FOR I = 1 TO 80
60 INPUT "STOCK CODE"; S$(I)
70 INPUT "QUANTITY"; M(I) : GOTO 300
72 IF M <> INT(M) THEN PRINT "MUST BE A POSITIVE INTEGER":
   GOTO 70
73 IF M<0 THEN PRINT "MUST BE A POSITIVE INTEGER":GOTO 70
74 IF F = 1 THEN PRINT "NO SUCH STOCK CODE" : GOTO 60
75 T1(I) = PC(I) X M(I)
76 T = T + T1(I)
80 IF S$(I) <> "000" THEN GOTO 60
84 IF S$(I) <> "BYE" THEN GOTO 1820
85 NEXT I
90 PRINT "TOTAL COST IS ";T;
100 INPUT "CASH";CA
105 PRINT
```

```
110 CG = CA -T
120 PRINT TAB(2);"HONG KONG SUPPLIES LITMITED"
130 PRINT
140 PRINT TAB(2);"STOCK CODE";TAB(17);"PRICE";TAB(27);
    "QUANTITY";TAB(34);"COST"
145 FOR I = 1 TO 80
150 IF S$(I)="000" THEN GOTO 180
160 PRINT TAB(2);S$(I);TAB(17);PC(I);TAB(27);M;TAB(34);T1(I)
170 NEXT I
180 PRINT TAB(2);"TOTAL";TAB(6);T
190 PRINT TAB(2);"CASH";TAB(6);CA
200 PRINT TAB(2);"CHANGE";TAB(6);CG
300 F=1
305 FOR I = 1 TO 80
310 IF SC$(I) = S$(I) THEN F = 0 : GOTO 75
320 NEXT I
330 GOTO 75
1820 END
```

Subject C - Female form 5 student

S: (Read problem Point-Of-Sale)

R: Tell me about the question.

S: Okay, start now ?

R: Yes

S: A customer go to a shop to buy stocks. Each stock has a number. It is sent to the computer to indicate which stock is to be brought. Then the quantity is also sent. Repeat to ask for stock code and quantity. When end, a "000" should be inputted. Tell total cost, ask for cash and print the bill. That's it.

R: Any special attention should be paid to those value to be inputted ?

S: For example, if the stock code inputted is not one in the data table, it should be rejected. The quantity inputted should be an integral value. If end of a transaction, then a "000" should be inputted. If cash inputted is not sufficient, ask for more. Finally, if no one buy stocks, then input "BYE" to end the program.

R: Any other supplements ?

S: 80 codes and price are given in the question. The price are given in one decimal place. "XXX" means end of data.

R: (Meaning of varaibles M and N given in the question are explained. N is used to store number of stocks in the shop.)

S: N is 80.

R: Why you are so sure that N is 80 ?

S: Because the question said that there are no more than 80 stock codes.

R: That's it, "no more than" 80 ! Not exactly 80.

S: How do I know the value of N ?

R: You have to find it by yourself. Have you encountered this case before, that is to count N until "XXX" is met ?

S: Yes.

R: Can you find N then ?

S: Use FOR/NEXT loop to count until it meets "XXX"... May be

no FOR/NEXT loop is needed. Set a variable to count the stock code until "XXX" is met.

R: Do you understand the question completely ?

S: Yes.

R: Now, you have to start to write the program and think aloud as well. Maybe you are used to think quietly but this time you have to try to speak up as you think.

R: You may start now.

S: The first job is to DIMension SC$(80).
(Write line 10)

S: Then input (Write line 20).
Pause

R: What are you thinking ?

S: Thinking is it necessary to add a bracket to the variable CSC$.

S: It is then necessary to search to see whether the code inputted is one in the data table. Inform the cashier when it is not found.

R: Have you write a search algorithm before ?

S: Yes, binary search.

R: Any other search ?

S: No, the other (she mean sequential search) is not necessary because it is seldom use.

R: You are requested to use sequential search in this question.

S: Okay, then use it.
Pause

R: What are you thinking ?

S: Thinking how to read data. I have to set a X$ to count the number of stock codes in the data table.
(She writes 30 N=1 ; 40 READ X$; 50 IF X$="XXX" ...)

S: I feel very uneasy when someone watching me to write a program.

R: Relax yourself. I just want to know how you think when you are creating a program. You may think quietly if you feel uncomfortable. Tell me about your thoughts with a short delay.

Pause

R: Are you trying to recall the past algorithm ?

S: No, my brain is empty.

R: Have you experienced this case before ?

S: Yes

R: How do you solve it ? By yourself or by the assisstance of the other ?

S: By myslef.

R: Can you solve it ?

S: Yes, it is solved in a short time.

Long pause

R: Are you thinking how to read ?

S: Ah... use N

R: What is the function of N ?

S: Count stock codes.

R: Can you try a FOR/NEXT loop ?

S: I have to find N. If a loop is used, N cannot be found.

R: Have you solved the reading problem without a FOR/NEXT loop ?

S: Use N = N + 1 .

R: When to add one to N ?

S: First, set N to 1. Read a data. If it is "XXX" then jump away else add 1 to N.

R: Yes, you are right.

(Subject try to write down the codes.)

Pause for writing codes in line 30, 40, 50 and 60.

R: You have finished reading, havn't you ?

S: The next job is to search to see whether the inputted stock code is one in the data table or not.

Long pause for writing codes in searching. (Write lines 65, 70, 80, 90, 100 and 110.)

R: Are you writing the codes in the way as you are taught ?

S: What are taught ?

R: This search.

S: I have forgotton.

R: What is the function of the "flag" ?

S: If "flag" equal to 1, that mean it is found. If it is 0,

that mean it is not found.
Pause

R: Now, the program lines for searching have been finished. What to do next ?

S: Ask for the quantity.
Pause for writing line 120.

S: Add...

R: Add what ?

S: Add line 25. If CSC$ equal to "000" then end the program. No, just go to a place where the cost will be calculated.

R: Are you validating the quantity ?
Pause for writing the codes on validating the quantity.

R: Have you written these codes before ?

S: Checking whether the value is an integral value ?

R: Yes

S: Yes, I have written similar codes before.
Pause for writing line 130.

R: What are you thinking ?

S: Ah... after the stock code and quantity are entered, I want to calculate the cost.
Pause for thinking.

S: Add line 15 FOR J = 1 to 80

R: What is the function of this FOR/NEXT loop ?

S: After one stock is inputted, try to see whether there is another to be inputted. If yes, continue. If no, all the stock codes and quantities are stored in separate locations using J as an indexing.
Long pause.

R: What are you thinking ?

S: Thinking about how to calculate the cost. Can I leave the cost behind until all stock codes and quantities are entered ?

R: Do you think so ?

S: May be...

R: Why bracket J is added to CSC$ on lines 20 and 25 ?

S: You have to identity the stock codes and quantities as 1, 2, 3, ... so that it can be retrieved for calculating the

cost in a later stage.

R: Why you DIMension it with 80 elements ?

S: There are no more than 80 stock codes, so it is defined as 80.

R: Why bracket J is added to Q in line 120 and line 130 ?

S: To ensure all quantities entered are in order.

R: In other words, every time the quantity entered are put into array and they are stored for later use.

S: For calculating the cost.

S: The price has not been taken out from the table yet...

S: This and that can calculate the cost. (She means Q(J) multiply PC(J) can produce cost.)

R: Does the end of the FOR/NEXT loop represent that you have finished input everything ?

S: May be.

R: Why you VALue Q(J) ?

S: Because Q(J) is a string. String cannot be used for calculation. (She add VAL to all Q(J) in line 130.)
Pause

R: What are you thinking ?

S: Calculate the total cost.
Pause

R: Are you thinking how to calculate ?

S: I want to find the price for calculating the cost.
Long pause (Looking into the question again.)

S: Can I tell you how I do this usually ? I have no idea to continue.

S: Usually this and that will be read out simultaneously (She means to read stock code and price form the data table). They are put into arrays.

R: So, do it.

S: No, I can't do it here. It is impossible because I don't know its position. I have just set the "flag" and only know whether it is present or not. I don't know the index.

R: You want to find the index.

S: If add one more line here (in sequential search), the index can be find. But I don't know how to do it yet. It can be

done here by writing it again, but it wastes time.

R: What do you mean by writing it again ? Do you mean to loop again here ? Read the data table again and find the price.

S: Find the first, then next and so on.

R: When data are read, they are read in order. You cannot read randomly.

S: That's why I cannot do it. But I still have no idea in solving this problem.

(Subject is entangled in finding price for each stock.)

R: Why not store the prices as you read the stock codes ? Is the purpose of the codes from line number 30 to 60 just for counting N ?

S: If I store the stock codes and prices...

Pause for thinking

R: Do you store them usually in similar cases ?

S: Yes

R: Store by what ?

S: Use array.

R: Why arrays are not used here ?

S: If I store them into array, and then find N...

I want to find price for a particular stock. I have to find the value of index "I".

R: If you know the value of "I", can you calculate the cost ?

S: Cost can be calculated.

Long pause for thinking how to find the value of "I".

S: (The subject add bracket K to SC$ and PC in line 40.)

R: What are you thinking now ?

S: Still cannot find the price.

R: How to continue if you can find I ?

S: If I can be found, then I shall find the cost...

Pause

(Subject write line 140 COST(J)=Q(J)XPC(J).)

R: Why array is used to store the cost ?

S: They will be used in a later stage.

Long pause

R: Are you calculating the total cost ?

S: (Subject writing line 150 T=0, 160 T = T + COST(J))

R: Are you thinking how to print ?

S: Becasue I have to print the total cost in a later stage, therefore I want to accumulate the cost in T. (However, the subject suspect whether the variable T can be added to an array variable COST( ).)
Pause

R: Are you thinking how to calculate the total cost ?

S: ...

R: What will you do if the total cost is calculated ?

S: I shall add NEXT J in line 170 to finish the loop. Print the total cost and ask for cash. Use a FOR/NEXT loop to print the bill. Finally print change by subtracting cash by total cost.

S: This program will then be finished, isn't it ?

S: Oh, I have missed a part. If CSC$(J) equal to "BYE", then the program will end.

R: Now, you only left calculating the total cost unsolved.

S: I have done it. But, I don't know whether it is correct or not. Is it possible to put a variable and an array variable in the same side of an assignment statement ?

R: Yes, it can.

S: After the total cost has been calculated, then print the total cost. If not enough cash, then ask for cash again. The next job is to use a FOR/NEXT loop to print all the required information in order because they are stored in arrays. The others follow the format of the bill as in the sample output.

R: Another problem that you have not solved yet is finding the position for price of a stock.

R: The possible solution is to ...

S: In fact, I have an idea to do it, but I don't know whether it can work or not. I remember when a "flag" is used, it is either "0" or "1". If "flag" can store the value of "I", then the problem will be solved. Can the "flag" store the value of "I" ?

R: Yes, that's right. You can do that, of course.

S: I don't know this can work becasue every time when a "flag"

is used it only indicates whether the work has been done or not. I have never used it this way.

S: But, if "flag" equal to 1, will it store N finally at the end of the looping ?

R: You can leave the loop when it is found.

S: We cannot jump out of loop.

R: You can use a variable to set I equal to N for leaving the loop.

S: Will the value of "I" changed then ?

R: You have set the value of "I" into the "flag". It will not be affected.

S: Yes, that's it. I have thought it this way. I do not know whether the "flag" can store the value of "I" or not.

R: In other words, you can find the price using this value in the "flag".

R: Another problem is in accumulating the total cost in lines 150 and 160. You have put them together too close. You set T equal to zero and then accumulate T immediately. T on the right hand side will be zero every time in line 160.

S: (Subject cancelled line 150 immediately.)

S: It should be put in the upper part of the program.

S: It's too short of time in setting this program so I make mistakes.

R: Are you doing program this way at home ?

S: Yes, I always make mistakes in the process of programming. I have to look backward very often. If more time is allowed, I shall look into the program lines in a more detail fashion. I shall spend much time in thinking when problems arise. Usually, I seldom skip the unsolved part and continue with the other parts like this time.

R: Do you use flowchart to assist your planning ?

S: Seldom, but I do use it for a certain part of a programming problem. Certainly no flowchart will be drawn for a whole problem.

R: Will you use flowchart in a question like the one you have just solved ?

S: Certainly not for the whole.

R: Which part then ?

S: More complicated part.

R: Which part is more complicated ?

S: In fact, this question is easy.

S: Many things are only mixed up.

S: I think I shall only draw flowchart for the part in reading data.

S: This program is, in fact, difficult to us in comparison to the assignments and examination questions. Usually, the steps are given in examinations. We only need to follow the steps. It will be more easy. But in this programming problem, we find mistakes at the rear part of the program. We have to jump back to make corrections. In examination, it is certainly correct if we follow the details of each part.

Programming Solution By Subject C

```
10 DIM SC$(80), CSC$(80), Q(80)
15 FOR J = 1 TO 80
20 INPUT "STOCK CODE ?"CSC$(J)
25 IF CSC$(J)="000" THEN GOTO ____
26 IF CSC$(J)="BYE" THEN END
27 FOR K = 1 TO 80
30 N = 1
40 READ SC$(K), PC(K)
50 IF SC$ = "XXX" THEN 65
60 N = N + 1
65 FLAG = 0
70 FOR I = 1 TO N
80 READ SC$, PC
90 IF SC$(I) = CSC$(I) THEN FLAG = I
100 NEXT I
110 IF FLAG = 0 THEN PRINT "NO SUCH STOCK CODE" : GOTO 20
120 INPUT "QUANTITY ?"Q(J)
130 IF VAL(Q(J)) < 0 OR VAL(Q(J)) - INT(VAL(Q(J))) <> 0 THEN
    PRINT "MUST BE A POSITIVE INTEGER" : GOTO 120
140 COST(J) = Q(J) X PC(J)
150 T = 0
160 T = T + COST(J)
170 NEXT J
180 ...
```

This is an incomplete programming solution because the interview runs out of time.